Ferienkurs Pascal

Joachim Goll und Johannes-Markus Waizenegger

Version 1.1

Inhaltsverzeichnis

1 Lektion 1 - Programme tippen 5

1.1 Ein erstes Programm 5

1.2 Die integrierte Entwicklungsumgebung IDE 5

1.2.1 Aufbau der Bedienoberfläche 5
1.2.2 Fenster 6
1.2.3 Editor 8
1.2.4 Arbeiten ohne Maus 8

1.3 Aufgaben 9

2 Lektion 2 - Programme in höheren Programmiersprachen 10

2.1 Einsatz von Programmen 10

2.2 Programmiersprachen der 1., 2. und 3. Generation 10

2.2.1 Maschinencode 10
2.2.2 Assembler 11
2.2.3 Programmiersprachen der 3. Generation 12
2.2.4 Zusammenfassung 13

3 Lektion 3 - Aufbau eines Pascal-Programms 14

3.1 Grammatik von Sprachen 15

3.2 Programmbeispiel 15

3.3 Allgemeiner Aufbau eines Programms 16

4 Lektion 4 - Variablen. Ein- und Ausgabe von Zahlen. 19

4.1 Variablen 19

4.2 Eingabe von der Tastatur 20

4.3 Aufgaben 21

5 Lektion 5 - Standarddatentypen in Pascal 22

5.1 Zeichen und Zeichensätze 22

5.2 Die Datentypen integer, real, boolean, char und string 28

5.2.1 Datentyp `integer` 28
5.2.2 Datentyp `real` 30
5.2.3 Datentyp `boolean` 31
5.2.4 Datentyp `char` 32
5.2.5 Zeichenketten 34
5.2.6 Datentyp `string` in Turbo Pascal 34
5.2.7 Typanpassung 35

5.3 Aufgaben 35

6 Lektion 6 - Anweisungen 39

6.1.1 Wertzuweisung 39
6.1.2 Verbundanweisung 40
6.1.3 Wiederholungsanweisungen 40
6.1.3.1 while-Anweisung 41
6.1.3.2 repeat- Anweisung 42
6.1.3.3 for-Anweisung 43
6.1.4 Fallunterscheidungen 44
6.1.4.1 if-Anweisung 44
6.1.4.2 case-Anweisung 46
6.1.5 Sprunganweisung 47

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

6.1.6 Leere Anweisung ... 47
6.2 Aufgaben ... 47
7 Lektion 7 - Elemente von Pascal ... 49
7.1.1 Zeichenvorrat ... 49
7.1.2 Symbole und Trenner ... 50
7.1.2.1 Trenner ... 50
7.1.3 Namen ... 54
8 Lektion 8 - Ausdrücke ... 56
8.1 Prioritäten der Operatoren ... 56
8.2 Arithmetische Ausdrücke ... 57
8.3 Logische Ausdrücke ... 58
8.4 Aufgaben ... 62
9 Lektion 9 - Ein- und Ausgabe ... 64
9.1.1 Textdateien und die Standarddateien `input` und `output` ... 64
9.1.2 Lesen von `input` ... 66
9.1.3 Standardfunktionen `eoln` und `eof` ... 67
9.1.4 Schreiben nach `output` ... 67
9.1.5 Beispiele für die Ein- und Ausgabe ... 69
10 Lektion 10 - Unterprogramme ... 71
10.1 Aufruf von Funktionen ... 72
10.2 Vereinbarung von Funktionen ... 73
10.3 Aufruf von Prozeduren ... 75
10.4 Die Direktive forward ... 81
10.5 Lokale und globale Variablen ... 83
10.6 Aufrufhierarchie und Unterprogrammreihenfolge ... 84
10.7 Aufgaben ... 86
11 Lektion 11 - Eindimensionale Arrays ... 91
11.1 Einfache und zusammengesetzte Datentypen ... 91
11.2 Der Arraytyp ... 92
11.3 Vereinbarung von Arrayvariablen für eine eindimensionales `integer`-Array ... 94
11.4 Arrays als Parameter bei Prozeduren ... 95
11.5 Aufgaben ... 95
12 Lektion 12 - Records ... 96
12.1 Definition eines Record-Typs ... 96
12.2 Vergleich Record- und Array-Struktur ... 99
13 Lektion 13 - Dateien ... 100
13.1 Schreiben und Lesen für Text-Dateien ... 101
13.2 Schreiben und Lesen für allgemeine Dateien ... 104
13.3 Aufgaben ... 108

Dieses Skript ist als Werk urheberrechtlich geschützt. Es darf nur mit Genehmigung der Autoren verbreitet werden.

Vorwort

Das vorliegende Skript wurde erstellt für den Ferienkurs "Pascal" im September 2000 an der FHTE Esslingen.

Es stellt eine Einführung in eine strukturierte Programmiersprache am Beispiel der **Programmiersprache Pascal** dar. Aus diesem Grund werden nicht alle Untiefen der Programmierung behandelt, sondern die Grundprinzipien der Programmierung erklärt, die sich auch leicht auch auf neuere Programmiersprachen wie C oder Java umsetzen lassen.

Pascal wird im Computer-Magazin[1] aus dem Jahre 1982 folgendermaßen beschreiben:

Keine andere Programmiersprache hat sich ohne Marketing-Promotion so rasch und so weit verbreitet wie Pascal. Eine einzige Veröffentlichung von Jensen und Wirth 1975 ("Pascal - User Manual and Report") gab die Initialzündung für beinahe weltweite Aktivitäten in Sachen Pascal bei Hard- und Softwareherstellern, Anwendern in Wirtschaft und Forschungseinrichtungen, in Schulen und nicht zuletzt im Personalcomputer-Bereich.

Die Intention von Wirth war:

- er wollte eine Sprache für den Unterricht schaffen, welche die in den wesentlichen Programmiersprachen durchgängig vorhandenen Konzepte enthielt, und dabei einfach zu erlernen sein sollte.
- er wollte eine Sprache schaffen, die keinen allzu komplexen Compiler erforderte und die sich leicht auf jedem Rechner implementieren ließ ("small is beautiful").

Auch wenn heute in der Industrie die Bedeutung von Pascal abnimmt und Anwendungen hauptsächlich in der Programmiersprache C, C++ oder Java geschrieben werden, ist die Sprache Pascal auch heute noch wegen ihrer Einfachheit die ideale Einstiegssprache in die Informatik.

[1] [Quelle: Willi Bolkart, Michael Löw, Die Programmiersprache Pascal, Computer Magazin 7, 1982, S. 49]

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

1 Lektion 1 - Programme tippen

1.1 Ein erstes Programm

Das erste **Programm** `begruessung` schreibt einen Satz auf den Bildschirm.

```
program begruessung;
begin
  write ('Hallo, ich begruesse Sie zum Pascal-Kurs.');
end.
```

Zum Schreiben eines solchen Programms braucht man einen Editor (lat. edire = hervorbringen). Ist das Programm geschrieben, so muss man es noch kompilieren und linken, damit es auf dem Prozessor ablaufen kann. Was Kompilieren und Linken ist, wird später noch erklärt.

Wichtig ist, dass man im Prinzip drei Programme zum Programmieren braucht:

- einen **Editor**
- einen **Compiler**
- einen **Linker**

Dies können drei unabhängige Werkzeuge sein, diese Werkzeuge können aber auch in ein einziges Werkzeug integriert sein und aus der Oberfläche dieses Werkzeugs heraus aufgerufen werden. Ein solches Werkzeug wird als IDE (Integrierte Entwicklungs-umgebung - Integrated Development Environment) bezeichnet.

Wenn man eine integrierte Entwicklungsumgebung benutzt, muss man sich nicht mehr so stark um die einzelnen Schritte wie Komplieren und Linken kümmern, da diese Aktionen per Knopfdruck ausgeführt werden.

In diesem Kurs wird die IDE des Borland Turbo Pascal Compilers benutzt.

1.2 Die integrierte Entwicklungsumgebung IDE

1.2.1 Aufbau der Bedienoberfläche

Der Bildschirm der integrierten Entwicklungsumgebung hat drei Teile:

- das **Auswahlmenü** in der obersten Zeile,
- die eigentliche **Arbeitsfläche**,
- und darunter die **Statuszeile**.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Menüzeile (Auswahlmenü)

Sämtliche Menübefehle sind über die Menüzeile erreichbar. Die Menüzeile ist immer sichtbar, es sei denn, auf dem Bildschirm werden gerade die Ausgaben eines Programms und nicht die IDE selbst dargestellt. Wenn das Auswahlmenü aktiviert ist, sieht man einen hervorgehobenen Menütitel. Dieser zeigt das aktuell gewählte Menü an.

Sämtliche Befehle lassen sich mit der Maus anklicken. Zur schnellen Anwahl von Befehlen ist es oftmals zweckmäßig, anstelle des Anwählens mit der Maus eine Anwahl mit Hilfe von Funktionstasten durchzuführen.

Die Menüzeile enthält die Befehle:

- File
- Edit
- Search
- Run
- Compile
- Debug
- Options
- Window
- Help

Arbeitsfläche

Unter der Menüzeile kommt die eigentliche Arbeitsfläche, die einen Rahmen hat.

Statuszeile

Unterhalb der Arbeitsfläche in der untersten Zeile der Bedienoberfläche kommt die Statuszeile.

Die Statuszeile dient für folgende Aufgaben:

- Sie meldet, was das Programm gerade macht. Beim Sichern einer Datei wird beispielsweise "Saving Dateiname..." ausgegeben.
- Sie gibt einzeilige Hinweise auf den gewählten Menü-Befehl und die sogenannten Wahlpunkte des Dialogfensters (Menüalternativen).
- Sie zeigt die aktuell verfügbaren Tastenkürzel für die Menübefehle. Diese können, anstatt im Menü gewählt zu werden, auch in der Statuszeile angeklickt werden.

1.2.2 Fenster

Das Arbeiten in der IDE erfolgt fast ausschließlich in Fenstern. Ein Fenster ist ein Teil der Bedienoberfläche, den man verschieben, vergrößern und verkleinern, teilen, mit anderen Fenstern verdecken, öffnen und schließen kann.

Man kann mehrere Fenster gleichzeitig öffnen. Aktiv ist allerdings immer nur ein einziges. Als **aktives Fenster** wird immer das Fenster bezeichnet, in dem man gerade arbeitet. Alle Kommandos und Texteingaben beziehen sich immer nur auf dieses

Fenster (Anmerkung: Öffnet man dieselbe Datei in mehreren Fenstern, so wirken sich Aktionen innerhalb eines Fensters auch auf die anderen Fenster aus).

Fast alle Fenster haben die folgenden Elemente gemeinsam:

- einen **Titel**
- ein **Schließfeld**
- einen **Rollbalken**
- ein **Eckfeld** zur Änderung der Größe
- ein **Zoomfeld**
- und eine **Fensternummer**

Das aktive Fenster erkennt man an der **doppelten Umrahmung**.

Das **Editfenster** zeigt zusätzlich die **aktuelle Zeilen- und Spaltennummern** in der linken unteren Ecke. Wurde die **Datei geändert**, so erkennt man dies an einem * links neben den Spalten- und Zeilennummern.

Das **Schließfeld** befindet sich in der oberen linken Ecke. Durch Anklicken des Schließfeldes kann man ein Fenster schließen. Ein Sonderfall ist das Hilfe-Fenster, welches mit ESC geschlossen wird oder durch Anklicken des Fensters an beliebiger Stelle.

Der **Titel**, der horizontale Balken unter der Menüzeile, enthält den Namen des Feldes und die Fensternummer. Durch ein einfaches Anklicken und Verschieben der Maus bei gedrückter Taste kann das **Fenster verschoben** werden. Durch doppeltes Anklicken kann man das Fenster auf volle Größe vergrößern bzw. die ursprüngliche Größe wieder herstellen.

Wird ein Fenster von einem anderen **Fenster verdeckt**, so kann man es durch die Tastenkombination Alt + Fensternummer wieder **in den Vordergrund bringen** oder durch Anklicken des Fensters mit der Maus.

Das **Zoomfeld** ist in der rechten oberen Ecke. Hat das Zoom-Feld einen Pfeil mit zwei Spitzen als Sinnbild, so hat das Fenster seine maximale Ausdehnung erreicht. Wird ein Pfeil nach oben als Sinnbild angezeigt, so kann man den Pfeil anklicken und das Fenster bis zu seiner maximalen Ausdehnung vergrößern.

Rollbalken sind horizontale bzw. vertikale Balken am rechten bzw. unteren Rahmen. Sie enthalten eine Positionsmarke, die die Position des Fensterausschnitts relativ zum Gesamttext anzeigt. Klickt man an einem Ende des Rollbalkens das Pfeilsymbol an, so wird um eine Zeile gerollt. Wird der schattierte Bereich an einer Seite des Rollbalkens angeklickt, so wird der Fensterinhalt um eine Bildschirmseite geblättert. Kontinuierliches Rollen erfolgt, wenn man die Maustaste gedrückt hält und die Positionsmarke zieht.

Das **Feld zum Verändern der Größe** ist in der rechten unteren Ecke. Wird diese Ecke bewegt, so wird das Fenster größer und kleiner.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Dialogfenster

Durch Anklicken von Menübefehlen mit der Maus wird ein Dialogfenster geöffnet.

Wird beispielsweise der Menübefehl File angeklickt, so öffnet sich das Pull-down Menue mit den Alternativen:

Open
New
Save
.
.
.
DOS shell
Exit

Mit dem Befehl **Open** öffnet man eine Datei, mit **Save** wird sie gesichert. Mit **Exit** kommt man auf die DOS-Ebene zurück (Ausstieg aus der integrierten Entwicklungsumgebung). Der Befehl **Dos shell** dient zur vorübergehenden Arbeit auf DOS-Ebene, um anschließend wieder in die integrierte Entwicklungsumgebung zurückzukehren.

1.2.3 Editor

Mit Hilfe der Tastenkombination Shift + Pfeiltasten wird Text markiert (oder durch Anklicken und Ziehen der Maus bei gedrückter Maustaste)

Mit Hilfe von Edit/Cut kann man den markierten Text aus der Datei entfernen und in das **Clipboard**, einem Zwischenspeicher, einfügen. Mit Paste kann man ihn in ein beliebiges Dokument an einer beliebigen Stelle einfügen. Der Text bleibt im Clipboard, so daß man ihn mehrfach verwenden kann. Mit Edit/Show Clipboard kann man das Clipboard-Fenster öffnen, das alle aus anderen Fenstern ausgeschnittenen und kopierten Textteile enthält. Hier kann man ein Textfragment markieren, das mit Edit/Paste eingefügt werden kann.

Durch Anwahl von Save aus dem File-Menü wird Text gespeichert (oder durch Drücken der Taste F2)

1.2.4 Arbeiten ohne Maus

In die Menüzeile kommt man durch Anklicken mit der Maus oder man gibt auf der Tastatur <F10> ein und bewegt sich in der Menüzeile mit den Pfeiltasten von Menübefehl zu Menübefehl weiter. Die Anwahl eines Befehls kann ohne Maus mit Hilfe der <RETURN>-Taste erfolgen. Im Pulldown-Menü bewegt man sich wieder mit Hilfe der Pfeiltasten und wählt mit <RETURN> einen Befehl an.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

1.3 Aufgaben

Aufgabe 1: Editieren mit Turbo Pascal

a) Schreiben Sie mit dem Editor von Turbo Pascal das folgende Programm:

```
program eins (input,output);

begin
    writeln ('hier ist Programm eins');
    write ('das war ');
    write ('es auch');
    writeln ('schon');
    writeln ('bye')
end.
```

b) Kompilieren und starten Sie das Programm. Welchen Unterschied zwischen `write` und `writeln` stellen Sie fest ?

c) Kehren Sie aus der integrierten Entwicklungsumgebung auf die DOS-Ebene zurück. Kopieren sie den File `Header.Pas` von `T:\TEMP\Goll` auf Ihr Verzeichnis. Rufen Sie die integrierte Entwicklungsumgebung wieder auf und fügen Sie mit Hilfe von Edit/Copy die Datei `Header.Pas` in Ihr Programm als Header ein. Füllen Sie den Header aus.

Aufgabe 2: Editieren mit Turbo Pascal

Schreiben Sie ein Programm, das die Wochentage Montag bis Sonntag jeweils in einer eigenen Zeile ausgibt.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

2 Lektion 2 - Programme in höheren Programmiersprachen

2.1 Einsatz von Programmen

Aufgabe 1:

Überlegen Sie sich, wozu Programme heuzutage Verwendung finden.

-

-

2.2 Programmiersprachen der 1., 2. und 3. Generation

Höhere Programmiersprachen werden in einer Sprache, die ähnlich zu Englisch ist, geschrieben. Es gibt Wörter wie `if`, `begin`, `end`, `while`, `repeat` etc.

Das war nicht immer so. Zu Beginn der Datenverarbeitung gab es noch keine höheren Programmiersprachen.

Grundsätzlich versteht ein Prozessor keine höheren Programmiersprachen, sondern nur sogenannte Maschinenbefehle. Maschinenbefehle sind Folgen aus Nullen und Einsen.

2.2.1 Maschinencode

[Quelle: Russell Rector-George Alexy, Das 8086/8088 Buch]

Ein Programm in **Maschinensprache (Objektcode)** ist eine **Folge von Binärzahlen**[2] wie z.B.:

00111010
01100000
00000000
01000111
00111010
01100001
00000000
10000000
00110010
01100010
00000000

[2] Binärzahlen haben als Ziffern nur Nullen oder Einsen

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Programmieren im Maschinensprache ist mit etlichen Schwierigkeiten verbunden:

- Die Programme beschreiben die Aufgabe, die der Computer ausführen soll, in einem für den Menschen nur schwer verständlichen Format.
- Der Programmierer macht häufig Fehler, da Binärzahlen einander sehr ähnlich sehen, insbesondere wenn man bereits einige Stunden mit ihnen gearbeitet hat. Fehler sind schwer zu entdecken.
- Die Eingabe der Programme ist sehr langwierig, da man jedes Bit[3] individuell bestimmen muß.

Der erste Fortschritt war Assembler.

2.2.2 Assembler

Eine offensichtliche Verbesserung des Programmierens stellt die Zuweisung einer Bezeichnung zu dem Befehlscode dar. Den Namen eines Befehlscodes nennt man ein **"Mnemonik"** oder mnemo-technisches Hilfsmittel. Die mnemotechnische Abkürzung eines Befehls sollte in irgendeiner Form beschreiben, was ein Befehl ausführt.

In der Tat liefert jeder Hersteller eines Prozessors einen **Satz von Mnemoniks für den Befehlssatz** s**eines Prozessors**. Sie sind Standard für einen gegebenen Prozessor. Das Problem bei der Auswahl von Mnemoniks für Befehle besteht darin, daß es nicht für alle Befehle anschauliche Kurzbezeichnungen gibt. Bei einigen Befehlen ist dies der Fall, sie bestehen aus offensichtlichen Abkürzungen z.B. SUB für Subtraktion, XOR für Exklusiv-ODER, während dies bei anderen überhaupt nicht der Fall ist. Daraus resultieren Mnemoniks wie etwa WMP, PCHL und SOB (versuchen Sie, ihre Bedeutung zu erraten!). Zusammen mit den Befehl-Mnemoniks wird ein Hersteller gewöhnlich auch den CPU-Registern[4] entsprechende Namen zuweisen. Ebenso wie bei den Befehlsnamen besitzen einige Register offensichtliche Bezeichnungen z.B. A für Akkumulator, während andere mehr historische Bedeutung besitzen.

Ein Programm mit den Standardbezeichnungen von Intel sieht z.B. folgendermaßen aus:

```
MOV AX,800H[5]
AND AX,BX
JNZ MARKE
MOV AX,1000H
```

Diese Darstellung ist wesentlich anschaulicher als ein Maschinencode. Man bezeichnet diese Schreibweise als **Assemblersprache**.

Die Übersetzung eines in Assemblersprache geschriebenen Quellprogramms in die Maschinensprache erfolgt mit Hilfe eines dafür zuständigen Programms, dem **Assembler**.

---

[3] Ein Bit kann den Wert 0 oder 1 annehmen

[4] Register sind schnelle Speicherstellen im Prozessor

[5] 800H ist eine Zahl nicht im Zehnersystem, sondern im Sechzehnersystem - die sogenannte Hexadezimaldarstellung

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Bei Benutzung des Wortes "Assembler" ist jeweils darauf zu achten, ob nun die **Assembler-Sprache** oder das **Assembler-Programm** (Übersetzungsprogramm) gemeint ist.

Nachteile des Assemblers

Beim Programmieren **in Assemblersprache** muss man eine sehr **detaillierte Kenntnis des verwendeten speziellen Prozessors** besitzen. Man muß wissen, welche Befehle und Register der Computer hat, wie die Befehle die verschiedenen Register beeinflussen, welche Adressier-Verfahren der Computer verwendet und weitere Informationen. Keine dieser Informationen ist für die Aufgabe, die der Computer letztlich ausführen muß, relevant.

Programme in Assemblersprache sind zwischen Computern verschiedener Hersteller **in der Regel nicht übertragbar,** da die Assemblersprache durch die Architektur des Computers bestimmt wird.

2.2.3 Programmiersprachen der 3. Generation

Die Lösung vieler der mit Assemblersprache-Programmen verbundenen Schwierigkeiten ist die Verwendung von "höheren" Sprachen der 3. Generation, von sogenannten "problemorientierten" Sprachen.

Derartige Sprachen gestatten die Beschreibung von Aufgaben in einer Art und Weise, die eher problemorientiert als computerorientiert ist. Jede Anweisung in einer höheren Programmiersprache führt eine erkennbare Funktion aus. Sie wird im allgemeinen mehreren Befehlen in Assemblersprache entsprechen.

Ein Programm, das **Compiler** oder Kompilierer genannt wird, übersetzt ein Quellprogramm in einer höheren Programmiersprache in Maschinencode (Objectcode).

Man kann grob gesprochen sagen, daß ein Programmierer ein Programm zehnmal schneller in einer höheren Sprache als in Assemblersprache schreiben kann. Dies betrifft nur das Schreiben des Programms, nicht die Definition der Aufgabe, Entwurf des Programms, Fehlersuche, Testen und Dokumentation.

Die Programmierer können sich auf ihre eigentliche Aufgabe konzentrieren. Sie brauchen nur noch geringe Hardware-Kenntnisse über den Computer, den sie programmieren, zu haben.

Dennoch kann es Fälle geben, wo der Einsatz von Assembler gegenüber höheren Programmiersprachen von Vorteil ist, wie z.B. bei der Ansteuerung von Hardware-Schnittstellen in Geräten.

Generell sollte man den Einsatz von Assembler auf kleine Programme beschränken, die entweder sehr zeitkritisch sind und eine hohe Effizienz erfordern, die bei höheren Programmiersprachen in diesem Maße nicht gegeben ist, oder bei der Ansteuerung von HW-Bausteinen in Geräten.

In allen anderen Fällen sollten höhere Programmiersprachen verwendet werden.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

2.2.4 Zusammenfassung

Ein Prozessor versteht nur Maschinenbefehle

Maschinenbefehle sind Folgen von Nullen und Einsen
z.B. 11110010

Befehle in der Programmiersprache Assembler enthalten Mnemonics wie z.B. MOV, Zahlenwerte und Speicherplätze

Ein Assembler genanntes Werkzeug übersetzt Programme, die in der Programmiersprache Assembler geschrieben sind, in Maschinencode.

Jeder Prozessor hat seinen speziellen Befehlssatz und damit seinen speziellen Assembler. In Assembler wird die HW-Struktur des Prozessors - seine Register - sichtbar.

Eine höhere Programmiersprache ist unabhängig vom speziellen Prozessor.

Ein in einer höheren Programmiersprache geschriebenes Programm wird durch einen Compiler in Maschinencode übersetzt.

3 Lektion 3 - Aufbau eines Pascal-Programms

Natürlich besteht ein **Programm** aus **Anweisungen an den Prozessor**, die dem Prozessor sagen, was er arbeiten soll. Anweisungen bearbeiten Daten. Also müssen auch **Daten** spezifiziert werden.

Der Aufbau eines Programms und seiner Bestandteile ist fest geregelt. Programmiersprachen sind **formale Sprachen** und haben eine strenge Grammatik. Beim Kompilieren prüft der Compiler, ob die Regeln der Programmiersprache eingehalten wurden. Nur wenn dies der Fall ist, kann der Compiler das Programm in Maschinencode übersetzen. Ansonsten kommen **Fehlermeldungen des Compilers**, an welchen Stellen im Programm er nicht zurecht kommt. Will der Programmierer sein Programm zum Laufen bringen, dann beginnt die Fehlersuche.

Der **Aufbau einer Anweisung** soll anhand eines Beispiels einer natürlichen Sprache verdeutlicht werden. In der deutschen Sprache besteht ein einfacher Satz formal aus:

```
Subjekt Prädikat Objekt Satzendezeichen
```

Beispiele dafür sind:

```
Ich lerne Pascal.
Tobias und Thomas spielen Trompete, Posaune und Waldhorn.
Marcel schreibt, liest, lobt und zerreißt Bücher.
```

Während bei natürlichen Sprachen die Satzordnung vertauscht werden kann, ist der Aufbau bei Computersprachen fest und darf nicht vertauscht werden.

Der natürliche Satz

```
Peter trinkt Kaffee.
```

kann umgestellt werden

```
Kaffee trinkt Peter.
```

und ist auch danach noch lesbar. Jedem ist auf Grund seiner Erfahrung klar, wer wen trinkt. Bei einem Computer gibt es keine Erfahrungen im Hintergrund. Nichts darf mehrdeutig sein. Alles muss streng geregelt sein.

Bei Computersprachen gibt es wie bei den natürlichen Sprachen **Grammatiken**, wie ein Programm und seine Bestandteile aufgebaut sein müssen.

3.1 Grammatik von Sprachen

Eine Sprache besteht aus aus richtig geschriebenen Wörtern. Zusätzlich haben diese Wörter auch eine Bedeutung. Die Lehre von der Rechtschreibung ist die **Syntax**. Die **Semantik** beschreibt den Sinn und die Bedeutung der Wortfolgen. Syntax und Semantik bilden die **Grammatik** einer Sprache.

Eine Grammatik umfasst **Syntax** und **Semantik**

Die **Syntax** der Sprache S ist die Definition der Menge aller in S **zulässigen Aussagen**.

Die **Semantik** der Sprache S ist die Definition der den zulässigen Aussagen (und nur diesen) zugeordneten **Bedeutungen**.

Syntaktisch falsche Sätze haben in keinem Fall eine Semantik, aber auch für syntaktisch korrekte Sätze ist sie nicht immer definiert. Beispiele dafür lassen sich leicht aus dem Bereich der natürlichen Sprachen finden, denen natürlich eine klare Definition der Grammatik fehlt. Man denke nur an Gedichte, die syntaktisch einwandfreie Sätze haben können, deren Bedeutung jedoch nicht eindeutig interpretiert werden kann.

Beispiel für eine Sprache aus der Mathematik:

Die natürlichen Zahlen, mit arabischen Ziffern im Dezimalsystem dargestellt, bilden eine einfache nicht-endliche formale Sprache.

Syntax:

Jede Zahl ist eine Sequenz von Ziffern (0..9), wobei die erste Ziffer nicht 0 ist.

Semantik:

Der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist.

3.2 Programmbeispiel

Zuerst soll an dem schon bekannten Beispiel-Programm `begruessung` der Aufbau eines Programms studiert werden, danach kommt dann zusammengefasst die allgemeingültige Theorie.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Das Programm begruessung

```
program begruessung (input, output);
begin
  write ('Hallo, ich begruesse Sie zum Pascal-Kurs')
end.
```

Die erste Zeile enthält den **Programmkopf** für das Programm `begruessung`. Im Programmkopf wird der **Programmname** angegeben.

In der 2. bis 4. Zeile befindet sich der **Anweisungsteil** - das **Hauptprogramm**. Es wird durch die beiden **Schlüsselwörter** `begin` und `end` begrenzt.

Das Hauptprogramm beinhaltet als einzige **Anweisung** die `write`-Anweisung mit der Buchstabenfolge

```
Hallo, ich begruesse Sie zum Pascal-Kurs
```

Das Programm wird mit einem Punkt beendet.

3.3 Allgemeiner Aufbau eines Programms

Ein Pascal-Programm besteht aus

- **Programmkopf**
- **Vereinbarungen**
- **Hauptprogramm**
- **Endepunkt**

Programmkopf

Um die Ein- und Ausgabe zu vereinfachen, gibt es die Standarddateien **`input`** und **`output`**. Im Normalfall wird über `input` gelesen und über `output` geschrieben. Normalerweise sind input und output die Tastatur und der Bildschirm.

Ein **Programmkopf** hat in der Regel die Form:

```
program test (input, output);
```

Beim Borland-Compiler kann `input` und `output` wegfallen:

```
program test;
```

Im **Vereinbarungsteil** werden die Daten, mit denen ein Programm etwas machen soll, durch sogenannte **Deklarationen** und **Definitionen** beschrieben. Ferner können dort Funktionen und **Prozeduren**, d.h. **Unterprogramme,** vereinbart werden.

Beachten Sie, dass der Begriff der Vereinbarung den Begriff der Definition und der Deklaration umfaßt.

Unterprogramme beinhalten öfter wiederkehrende Anweisungsfolgen. Durch die Einführung von Unterprogrammen soll das eigentliche Hauptprogramm nicht zu groß werden und damit übersichtlich bleiben. Unterprogramme können mehrfach aufgerufen, d.h. wiederverwendet werden.

Unterprogramme werden bei kleinen Programmen nicht eingesetzt. Hier begnügt man sich mit dem Hauptprogramm.

Das **Hauptprogramm (Anweisungsteil)** wird beim Programmstart ausgeführt. Der **Anweisungsteil** enthält die Aktionen, die mit den im Vereinbarungsteil charakterisierten Daten durchgeführt werden sollen, in Form von sogenannten **Anweisungen** (**statements**).

Das Hauptprogramm wird durch die beiden Schlüsselwörter `begin` und `end` begrenzt. Dazwischen können beliebig viele Anweisungen stehen.

Das Hauptprogramm besteht aus:

```
begin
   Anweisungen
end
```

Die der Reihe nach aufgeschriebenen Anweisungen werden jeweils durch einen Strichpunkt getrennt.

Der Vereinbarungsteil soll an dieser Stelle übersprungen werden. Auf die einzelnen Elemente soll bei ihrer ersten Verwendung eingegangen werden. Nur zum Nachschlagen wird er hier beschrieben:

Im Vereinbarungsteil kommen

an 1. Stelle: Markenvereinbarungen (**Marken-Deklarationsteil**), wenn Marken verwendet werden
an 2. Stelle: Konstantendefinitionen (**Konstanten-Definitionsteil**), wenn Konstanten verwendet werden

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

an 3. Stelle: Typdefinitionen (**Typ-Definitionsteil**), wenn eigene Datentypen eingeführt werden
an 4. Stelle: Variablenvereinbarung (**Variablen-Deklarationsteil**)
anschließend: Prozedurvereinbarungen bzw. Funktionsvereinbarungen (**Prozedur- und Funktions-Deklarationsteil**)
Anmerkung: Hierbei ist die Reihenfolge frei, d.h. Funktionen und Prozeduren können in beliebiger Reihenfolge vereinbart werden.

Mit der **Markenvereinbarung** werden mögliche Sprungziele für Sprunganweisungen im Anweisungsteil beschrieben.

Mit der **Konstantendefinition** werden Konstanten mit einem Namen belegt und erhalten einen Wert. Im Programm werden nur die Konstantennamen verwendet. Damit erfolgen eventuelle Änderungen nur an einer zentralen Stelle.

Typdefinitionen erlauben es, eigene Datentypen entsprechend der jeweiligen Anwendung zu definieren.

Bei der **Variablenvereinbarung** sind alle später im Anweisungsteil des Blockes vorkommenden Variablen mit Datentyp und Namen festzulegen.

Prozeduren und **Funktionen** sind **Unterprogramme**. Sie dienen dazu, Programmteile als eigene Programmeinheit mit eigenem Namen festzulegen. Mit diesem Namen können diese Programmeinheiten im Programm aufgerufen werden.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

4 Lektion 4 - Variablen. Ein- und Ausgabe von Zahlen.

Ausgaben auf den Bildschirm haben wir bereits erzeugt. Dabei hatten wir der **write-Prozedur** eine Zeichenkette übergeben, den diese auf den Bildschirm ausgab. Wenn wir jetzt von der Tastatur beispielsweise eine Zahl an das Programm geben wollen und diese Zahl zu einem späteren Zeitpunkt wieder abholen wollen, so muss das Programm unsere Eingabe aufheben (speichern). Die Speicherung erfolgt im sogenannten Arbeitsspeicher des Computers.

4.1 Variablen

Nach dem **Variablenkonzept** werden zur Speicherung Variablen verwendet. Variablen sind nichts anderes als **Speicherplätze im Arbeitsspeicher** (Memory), die vom Programmierer **über einen Namen angesprochen** werden können. Der Compiler legt aber einen solchen Speicherplatz nur an, wenn man ihn darum bittet. Diese Bitte muss im Vereinbarungsteil ausgesprochen werden. Hier gleich ein Beispiel:

```
program variable;
```
Programmkopf

```
var a: integer;
    b: real;
```
Vereinbarungsteil

```
begin
  a:= 5;
  b:= 1.5;
  writeln (a, b);
  a: = 3;
  writeln (a, b);
end.
```
Anweisungsteil

Das Schlüsselwort

`var`

kennzeichnet im Vereinbarungsteil die Definition von Variablen. Kommt der Compiler an diese Stelle, so sieht er, dass er die Variablen mit den Namen `a` und `b` im Arbeitsspeicher anlegen muss.

Der Compiler will aber natürlich wissen, von welchem Typ denn die Variable sein soll. Denn eine ganze Zahl (Typ `integer`) wird ganz anders gespeichert als eine reelle Zahl, auch Gleitkommazahl genannt (Typ `real`).

Eine Vereinbarung kann mehrere Variablen desselben Typs getrennt durch Kommas enthalten:

```
var a, b : integer;
```

Diese Vereinbarung ist äquivalent zu

```
var a: integer;
    b: integer;
```

Variablen liegen während der Programmausführung in Speicherzellen des Arbeitsspeichers. Die Speicherzellen des Arbeitsspeichers (siehe Bild 4-1) sind durchnummeriert. Beim PC ist eine Speicherzelle 1 Byte groß. Die Nummern der Speicherzellen werden **Adressen** genannt. Eine Variable kann natürlich mehrere Speicherzellen einnehmen.

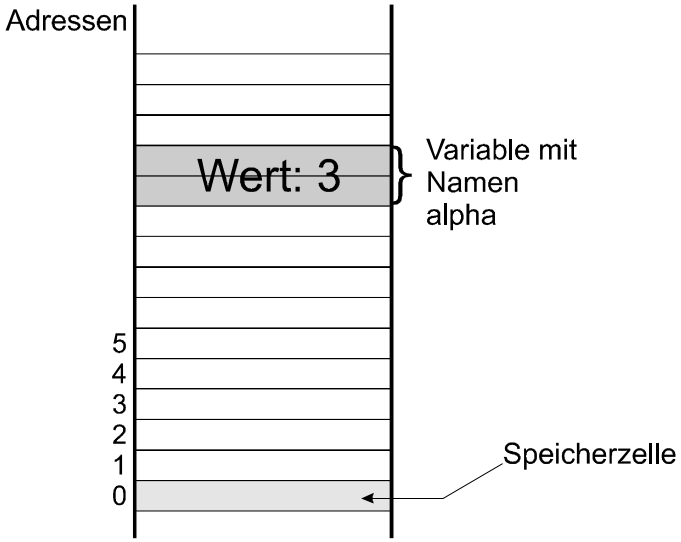


Bild 4-1Variable im Arbeitsspeicher

Mit Hilfe des **Zuweisungsoperators**

```
:=
```

wird der Variablen a in obigem Programm einmal der Wert 5 und einmal der Wert 3 zugewiesen. Die Variable lässt das problemlos mit sich machen. Sie ist ja flexibel (variabel). Bei einer Konstanten hätten wir hiermit kein Glück. Eine Konstante ist standhaft und behält konstant den Wert bei, mit dem sie versehen wurde.

4.2 Eingabe von der Tastatur

Wenn Ausgeben in Pascal `write` heisst, ist es nicht verwunderlich, dass das Einlesen mit der `read`-Prozedur geht. Auch hier gleich wieder ein Beispiel:

```
program einlesen;

var a: integer;
    b: real;

begin
  read (a);
  read (b);
  write (a, b);
end.
```

Bei der Prozedur `read` erwartet das Programm, dass Sie eine ganze Zahl eingeben. Machen Sie keine Eingabe, so macht das Programm einfach nicht weiter. Eine 3 etwa müssen Sie eingeben mit

3 <CR>

Das Symbol <CR>[6]

bewirkt, dass die 3 von der Tastatur an das Programm rübergeschoben wird. Der Zeilenvorschub <CR> wird mit der Returntaste erzeugt.

4.3 Aufgaben

Aufgabe 1: Standard-Prozeduren read und write. Ausdrücke

Berechnen Sie die Summe, Differenz, das Produkt und den Quotienten von 2 eingegebenen ganzen Zahlen. Verwenden Sie hierfür die Operatoren `+, -, *` und `/`

Aufgabe 2: Standard-Prozeduren read und write. Ausdrücke

Berechnen Sie die Summe, Differenz, das Produkt und den Quotienten von 2 eingegeben reellen Zahlen.

Aufgabe 3: Standard-Prozeduren read und write. Ausdrücke

Berechnen Sie von einem Rechteck die Fläche und den Umfang. Die Seitenlängen werden über die Tastatur eingegeben. Die notwendigen Formeln sind:

Flaeche = a * b
Umfang = 2 * (a + b)

[6] CR heisst Carriage Return, auf deutsch Wagenrücklauf

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

5 Lektion 5 - Standarddatentypen in Pascal

Zu den Standarddatentypen in Pascal gehört unter anderem auch der Typ `char`. Das Wort `char` kommt von character (Zeichen). In Kap 5.1soll zunächst erklärt werden, was ein Zeichen und ein Zeichensatz ist. Anschließend wollen wir uns den anderen Datentypen zuwenden.

5.1 Zeichen und Zeichensätze

Ein **Zeichen** ist ein von anderen Zeichen unterscheidbares Objekt, welches in einem bestimmten Zusammenhang eine definierte Bedeutung trägt. Solche Zeichen können beispielsweise Symbole, Bilder oder Töne sein.

Zeichen derselben Art sind **Elemente** eines **Zeichenvorrats**.

So sind beispielsweise die Zeichen I, V, X, L, C, M Elemente des Zeichenvorrats der römischen Zahlen.

Eine **Ziffer** ist ein Zeichen, das die Bedeutung einer Zahl hat.

Von einem **Alphabet** spricht man, wenn der Zeichenvorrat eine strenge Ordnung aufweist.

So stellt beispielsweise die geordnete Folge der Elemente

0, 1	das Binäralphabet,
a, b, c, ..., z	die Kleinbuchstaben ohne Umlaute und ohne ß,
0, 1, ..., 9	das Dezimalalphabet

dar.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Rechnerinterne Darstellung von Zeichen

Zeichen sind zunächst **Buchstaben** wie z.B. der Buchstabe a, **Ziffern** wie z.B. die Ziffer 3 oder **Sonderzeichen wie z.B.** das Malzeichen *. Zu diesen Zeichen können auch noch **Steuerzeichen**[7] hinzukommen.

Zeichen können die Bedeutung haben von:
- Buchstaben
- Ziffern
- Sonderzeichen
- Steuerzeichen

Rechnerintern werden die Zeichen durch Bits dargestellt. Ein **Bit** kann den Wert 0 oder 1 annehmen. Das bedeutet, dass man mit einem Bit 2 verschiedene Fälle darstellen kann. Mit einer Gruppe von 2 Bits hat man 2 * 2 = 4 Möglichkeiten, mit einer Gruppe von 3 Bits kann man 2 * 2 * 2 = 8 verschiedene Fälle darstellen, und so fort. Mit 3 Bits sind die Kombinationen

000, 001, 010, 011, 100, 101, 110 und 111

möglich. Die Umrechnung von Zahlen in der Binärdarstellung in die Dezimaldarstellung erfolgt über die **Stellenwerttabelle** wie in folgendem Beispiel der Binärzahl 10111:

Stellenwerttabelle

2^4	2^3	2^2	2^1	2^0
1	0	1	1	1

Die in der Stellenwerttabelle eingetragene Zahl lautet:

$1 * 2^4 + 0 * 2^3 \quad + 1 * 2^2 \quad + 1 * 2^1 \quad + 1 * 2^0$

$= 16 \quad + 0 \quad + 4 \quad + 2 \quad + 1$

= 23 in Dezimaldarstellung

Jeder dieser Bitgruppen kann man nun je ein Zeichen zuordnen. Das heißt, jede dieser Bitkombinationen kann ein Zeichen repräsentieren. Man braucht nur eine **eindeutig umkehrbare Zuordnung** (z.B. erzeugt durch eine Tabelle) und dann kann man umkehrbar eindeutig jedem Zeichen eine Bitkombination und jeder Bitkombination ein Zeichen zuordnen. Mit anderen Worten, man bildet die Elemente eines Zeichenvorrats auf die Elemente eines anderen Zeichenvorrats ab. Diese Abbildung bezeichnet man als **Codierung**.

[7] **Steuerzeichen** sind Zeichen, die für die Ansteuerung von Peripheriegeräten - wie z.B. einem Drucker - oder zur Steuerung einer rechnergestützten Datenübertragung dienen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Der Rechner arbeitet intern mit Zahlen. Es gibt eine eindeutige Zuordnung zwischen Nummern und Zeichen

Rechnerintern werden Zahlen durch Bits dargestellt.

Begriff eines Codes

Ein **Code** eine Vorschrift für die **eindeutige Zuordnung** oder **Abbildung** der Zeichen eines Zeichenvorrats zu denjenigen eines anderen Zeichenvorrats, der sogenannten Bildmenge. Der Begriff des Codes ist nicht eindeutig. Oftmals wird unter **Code** auch der **Zeichenvorrat der Bildmenge** verstanden.

Relevante Codes für Rechner

Für die Codierung von Zeichen im Binäralphabet gibt es viele Möglichkeiten, z.B. die Darstellung mit Hilfe von 7, 8 oder 16 Bits. Für Rechner besonders relevant sind Codes, die ein Zeichen durch 7 bzw. 8 Bits repräsentieren. Mit 7 Bits kann man 128 verschiedene Zeichen codieren, mit 8 Bits 256 Zeichen. Zu den am häufigsten verwendeten Zeichensätzen gehören:

- der ASCII[8]-Zeichensatz mit 128 Zeichen - eine nationale Variante des ISO-7-Bit-Code (ISO 646) in den USA, die aber weit verbreitet ist,
- der erweiterte ASCII-Zeichensatz mit 256 Zeichen

Der ASCII-Zeichensatz

Der ASCII-Zeichensatz ist die US-nationale Ausprägung des ISO-7-Bit-Codes (ISO 646). Eine weitere nationale Ausprägung des ISO-7-Bit-Codes ist der nach DIN 66003 spezifizierte deutsche Zeichensatz, bei dem die Zeichen Ä, Ö, Ü, ä, ö, ü und ß berücksichtigt wurden. Im DIN-Zeichensatz sind gegenüber dem ASCII-Zeichensatz folgende Änderungen vorgenommen worden:

```
[=Ä   \=Ö   ]=Ü   {=ä   |=ö   }=ü   ~=ß
```

Bei manchen Rechnern wie z.B. beim PC wird aber ein erweiterter ASCII-Zeichensatz eingesetzt, bei dem alle 8 Bits verwendet werden. Die ersten 128 Zeichen stimmen dabei mit dem 7-Bit ASCII-Code in Bild 5-1überein. Die Sonderzeichen Ä, Ö, Ü, ä, ö, ü und ß befinden sich dabei im Bereich 128-255.

Wie aus Bild 5-1und Bild 5-3 ersichtlich ist, enthält der ASCII-Zeichensatz Buchstaben, Ziffern, Sonderzeichen und Steuerzeichen. Da jedem Zeichen im Rahmen des jeweiligen Codes eine 7- bzw. 8-stellige Binärzahl eindeutig zugeordnet ist und die

[8] **ASCII** = American Standard Code for Information Interchange. Genauere Information zum ASCII-Zeichensatz ist in Anhang A zu finden.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Binärzahlen selbst eine geordnete Menge darstellen, bilden damit in jedem dieser Codes die Zeichen eine geordnete Menge. Es gibt für die Zeichen also ein vorher (<) und nachher (>), so dass die Zeichen dem Code entsprechend alphabetisch sortiert werden können.

Dez.	Hex.	Ctrl-Ch.	Char.	Dez.	Hex.	Char.	Dez.	Hex.	Char.	Dez.	Hex.	Char.
0	00	^@		32	20		64	40	@	96	60	‘
1	01	^A	□	33	21	!	65	41	A	97	61	a
2	02	^B	□	34	22	"	66	42	B	98	62	b
3	03	^C	□	35	23	#	67	43	C	99	63	c
4	04	^D	□	36	24	$	68	44	D	100	64	d
5	05	^E	□	37	25	%	69	45	E	101	65	e
6	06	^F	□	38	26	&	70	46	F	102	66	f
7	07	^G	□	39	27	’	71	47	G	103	67	g
8	08	^H	□	40	28	(	72	48	H	104	68	h
9	09	^I	□	41	29	)	73	49	I	105	69	i
10	0A	^J	□	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	□	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	□	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	□	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	□	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	□	47	2F	/	79	4F	O	111	6F	o
16	10	^P	□	48	30	0	80	50	P	112	70	p
17	11	^Q	□	49	31	1	81	51	Q	113	71	q
18	12	^R	□	50	32	2	82	52	R	114	72	r
19	13	^S	□	51	33	3	83	53	S	115	73	s
20	14	^T	¶	52	34	4	84	54	T	116	74	t
21	15	^U	§	53	35	5	85	55	U	117	75	u
22	16	^V	□	54	36	6	86	56	V	118	76	v
23	17	^W	↨	55	37	7	87	57	W	119	77	w
24	18	^X	□	56	38	8	88	58	X	120	78	x
25	19	^Y	□	57	39	9	89	59	Y	121	79	y
26	1A	^Z	□	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	□	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	□	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	□	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	□	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	□	63	3F	?	95	5F	_	127	7F	□

Bild 5-1Der ASCII-Zeichensatz mit 128 Zeichen

Wie man sieht, beginnen die Großbuchstaben des Alphabets bei 65, die Kleinbuchstaben bei 97. Der Abstand zwischen einem Groß- und dem dazugehörigen Kleinbuchstaben ist immer 32, einer Zweierpotenz. Bei der bitweisen Betrachtung stellt man die Zahl als Folge von Nullen und Einsen dar. ("Zweier-System"). Hier erkennt man, dass sich Groß- und Kleinbuchstaben nur in einem Bit unterscheiden.

Buchstabe A = 65 = 01**0**0 0001 binär (64 + 1)
Buchstabe a = 97 = 01**1**0 0001 binär (64 + 32 + 1)

In der Spalte Control-Character (Steuerzeichen), abgekürzt durch Ctrl-Ch., werden spezielle Tastenkombinationen angegeben, mit denen Steuerzeichen erzeugt werden können. Hierbei kann es je nach Betriebssystem auch geringfügige Modifikationen geben. Die ersten 32 ASCII-Zeichen stellen Steuerzeichen für die Ansteuerung von

Peripheriegeräten und die Steuerung einer rechnergestützten Datenübertragung dar. Diese Zeichen tragen die Namen:

Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol
0	NUL	6	ACK	12	FF	18	DC2	24	CAN	30	RS
1	SOH	7	BEL	13	CR	19	DC3	25	EM	31	US
2	STX	8	BS	14	SO	20	DC4	26	SUB		
3	ETX	9	HAT	15	SI	21	NAK	27	ESC		
4	EOT	10	LF	16	DLE	22	SYN	28	FS		
5	ENQ	11	VT	17	DC1	23	ETB	29	GS		

Bild 5-2 Namen der 32 Steuerzeichen des ASCII-Zeichensatzes

So ist beispielsweise FF die Abkürzung für Form Feed, d.h. Seitenvorschub, oder **CR** die Abkürzung für **Carriage Return**, den **Wagenrücklauf**[9]**.**

[9] Das Wort Wagenrücklauf kommt vom Wagenrücklauf der Schreibmaschine.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char
0	00		43	2B	+	86	56	V	129	81	ü	172	AC	¼	215	D7	╫
1	01	□	44	2C	,	87	57	W	130	82	é	173	AD	¡	216	D8	┘
2	02	□	45	2D	-	88	58	X	131	83	â	174	AE	«	217	D9	┘
3	03	□	46	2E	.	89	59	Y	132	84	ä	175	AF	»	218	DA	┘
4	04	□	47	2F	/	90	5A	Z	133	85	à	176	B0	□	219	DB	┘
5	05	□	48	30	0	91	5B	[	134	86	å	177	B1	□	220	DC	┘
6	06	□	49	31	1	92	5C	\	135	87	ç	178	B2	□	221	DD	┘
7	07	□	50	32	2	93	5D	]	136	88	ê	179	B3	□	222	DE	┘
8	08	□	51	33	3	94	5E	^	137	89	ë	180	B4	□	223	DF	┘
9	09	□	52	34	4	95	5F	_	138	8A	è	181	B5	□	224	E0	α
10	0A	□	53	35	5	96	60	'	139	8B	ï	182	B6	□	225	E1	β
11	0B	□	54	36	6	97	61	a	140	8C	î	183	B7	□	226	E2	Γ
12	0C	□	55	37	7	98	62	b	141	8D	ì	184	B8	□	227	E3	π
13	0D	□	56	38	8	99	63	c	142	8E	Ä	185	B9	□	228	E4	Σ
14	0E	□	57	39	9	100	64	d	143	8F	Å	186	BA	□	229	E5	σ
15	0F	□	58	3A	:	101	65	e	144	90	É	187	BB	□	230	E6	µ
16	10	□	59	3B	;	102	66	f	145	91	æ	188	BC	□	231	E7	τ
17	11	□	60	3C	<	103	67	g	146	92	Æ	189	BD	□	232	E8	ϕ
18	12	□	61	3D	=	104	68	h	147	93	ô	190	BE	□	233	E9	θ
19	13	□	62	3E	>	105	69	i	148	94	ö	191	BF	□	234	EA	Ω
20	14	¶	63	3F	?	106	6A	j	149	95	ò	192	C0	□	235	EB	δ
21	15	§	64	40	@	107	6B	k	150	96	û	193	C1	□	236	EC	∝
22	16	□	65	41	A	108	6C	l	151	97	ù	194	C2	□	237	ED	∅
23	17	↨	66	42	B	109	6D	m	152	98	ÿ	195	C3	□	238	EE	ε
24	18	□	67	43	C	110	6E	n	153	99	Ö	196	C4	□	239	EF	∩
25	19	□	68	44	D	111	6F	o	154	9A	Ü	197	C5	□	240	F0	≡
26	1A	□	69	45	E	112	70	p	155	9B	¢	198	C6	□	241	F1	±
27	1B	□	70	46	F	113	71	q	156	9C	£	199	C7	□	242	F2	≥
28	1C	□	71	47	G	114	72	r	157	9D	¥	200	C8	□	243	F3	≤
29	1D	□	72	48	H	115	73	s	158	9E	□	201	C9	□	244	F4	⌠
30	1E	□	73	49	I	116	74	t	159	9F	ƒ	202	CA	□	245	F5	
31	1F	□	74	4A	J	117	75	u	160	A0	á	203	CB	□	246	F6	÷
32	20		75	4B	K	118	76	v	161	A1	í	204	CC	□	247	F7	≈
33	21	!	76	4C	L	119	77	w	162	A2	ó	205	CD	□	248	F8	□
34	22	"	77	4D	M	120	78	x	163	A3	ú	206	CE	□	249	F9	•
35	23	#	78	4E	N	121	79	y	164	A4	ñ	207	CF	□	250	FA	·
36	24	$	79	4F	O	122	7A	z	165	A5	Ñ	208	D0	□	251	FB	√
37	25	%	80	50	P	123	7B	{	166	A6	ª	209	D1	□	252	FC	□
38	26	&	81	51	Q	124	7C	|	167	A7	º	210	D2	□	253	FD	²
39	27	’	82	52	R	125	7D	}	168	A8	¿	211	D3	□	254	FE	□
40	28	(	83	53	S	126	7E	~	169	A9	□	212	D4	□	255	FF	
41	29	)	84	54	T	127	7F	□	170	AA	¬	213	D5	□			
42	2A	*	85	55	U	128	80	Ç	171	AB	½	214	D6	□			

Bild 5-3 Der erweiterte ASCII-Zeichensatz (256 Zeichen)

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

5.2 Die Datentypen integer, real, boolean, char und string

In Pascal sind standardmäßig[10] die folgenden Datentypen vorhanden:

```
integer
real
boolean
char
```

In Turbo Pascal kommt u.a. der Datentyp

`string`[11]

für Stringvariablen dazu.

Der Benutzer kann diese Namen in der vorgesehenen Bedeutung ohne weitere Maßnahmen verwenden.

5.2.1 Datentyp `integer`

[Quelle: J. Ludewig, Einführung in die Informatik]

Der Datentyp `integer` ist eine echte (weil endliche) Teilmenge der ganzen Zahlen. Er vertritt in unseren Programmen die ganzen Zahlen. Jeder Computer hat nur einen endlichen Zahlenbereich, der nicht überschritten werden kann

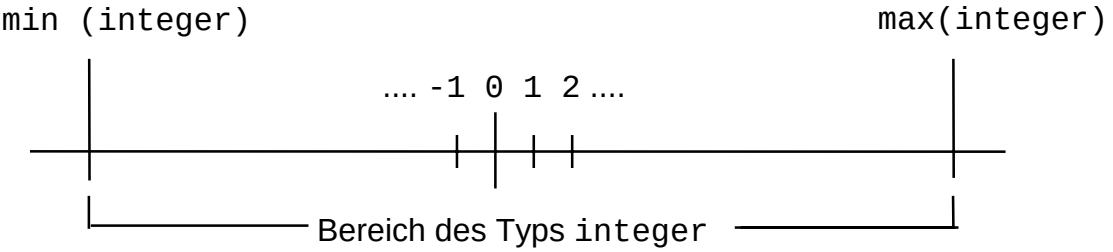


Bild 5-4 `integer` *-Zahlen*

min(integer) und max(integer) seien die Grenzen des Zahlenbereichs auf einer Maschine. Somit gilt für jede beliebige integer-Zahl x:

x ist eine ganze Zahl, min(integer) ≤ x ≤ max(integer)

Die größte positive ganze Zahl eines Computers steht in Pascal als Standard-Konstante mit dem Namen `maxint` zur Verfügung.

[10] man kann auch Datentypen selbst definieren wie Aufzählungstypen und Recordtypen
[11] Der Datentyp `string` ist nicht in Standard-Pascal enthalten. Er gehört zu Turbo-Pascal.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Das folgende Beispiel zeigt, daß man den Unterschied zwischen integer-Zahlen und ganzen Zahlen nicht vergessen darf:

```
program doppelt (input, output);
var x, zwei_x : integer;

begin
   read (x);
   zwei_x:= 2 * x;
   writeln (zwei_x)
end.
```

Die Zahl x darf nicht jeden Wert aus ihrem Wertebereich annehmen. Ist 2 * x größer als max(integer) oder kleiner als min(integer), so kommt es bei der Multiplikation zum **Zahlenüberlauf** und zu Fehlern. In den Übungen wird auf dieses Phänomen eingegangen.

Die folgende Tabelle zeigt den Wertebereich für den Datentyp `integer` und den Datentyp `longint`. Der Datentyp `longint` ist Turbo Pascal spezifisch und nicht im Standard-Pascal[12] enthalten.

Typ	Wertebereich	Größe in bit	Größe in Byte[13]
`Integer`	-32768 .. 32767	16	2
`Longint`	-214748648 .. 214748647	32	4

Muss man mit großen ganzen Zahlen arbeiten, so wird man den Datentyp `longint` verwenden.

Integer-Konstanten

Die Variablen vom Typ `integer` haben als Werte ganze Zahlen.

Die Konstanten des Typs `integer` sind positive oder negative ganze Zahlen (einschließlich der Null). Ein negatives Vorzeichen muß, ein positives kann geschrieben werden. Beispiele für ganze Zahlen sind:

```
  89   -105   +6
```

Zweierkomplement

Ganze Zahlen werden meist im sogenannten **Zweierkomplement** gespeichert. Zur Erläuterung soll folgendes Beispiel einer Zweierkomplement-Zahl von der Größe 1 Byte dienen:

[12] Standard-Pascal ist nach ISO normiert
[13] Ein Byte umfaßt eine Gruppe von 8 Bits

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Bitmuster

1	0	1	0	0	1	1	1

Stellen-wertigkeit -2^7 $+2^6$ $+2^5$ $+2^4$ $+2^3$ $+2^2$ $+2^1$ $+2^0$

Der Wert dieses Bitmusters errechnet sich aufgrund der Stellenwertigkeit zu:

$-1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 =$
$-128 + 32 + 4 + 2 + 1 = -89$

Das höchste Bit der Zweierkomplement-Zahl gibt das Vorzeichen an. Ist es Null, so ist die Zahl positiv, ist es 1, so ist die Zahl negativ.

Die dem Betrag nach größte positive Zahl in dieser Darstellung ist:

$(0111\ 1111)_2 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$

Die dem Betrag nach größte negative Zahl in dieser Darstellung ist:

$(1000\ 0000)_2 = -128$

Die tiefgestellte 2 bedeutet, dass es sich bei der Zahl um ein Bitmuster, welches bekanntlich die Basis 2 hat, handelt.

Addiert der Programmierer zur größten positiven Zahl eine 1 dazu, so befindet er sich bei der Zweierkomplementdarstellung plötzlich im negativen Zahlenbereich

5.2.2 Datentyp `real`

Zahlen vom Typ `real` sind eine endliche Menge rationaler Zahlen, die im Computer die rationalen und reellen Zahlen der Mathematik vertreten. Der Wertebereich ist also endlich, und die Genauigkeit der Darstellung einer rationalen oder reellen Zahl ist begrenzt.

Typisch sind im Rechner Zahlen der Form $\pm n * 2^{(\pm m)}$ exakt darstellbar, wobei m und n Integer-Zahlen sind. Dagegen sind nicht nur die transzendenten und die übrigen nicht-rationalen Zahlen (z.B. e, Quadratwurzel aus 2) ausgeschlossen, sondern auch Brüche wie 1/3 und sogar Dezimalbrüche wie 0,1! Außerdem ist die Rechner-Arithmetik mit `real`-Zahlen Rundungsfehlern unterworfen und gehorcht daher auch im Bereich der darstellbaren Zahlen nicht den Gesetzen der Mathematik! Beispielsweise ist im allgemeinen das Ergebnis eines Rechners für A + B - B, wobei A und B positive Real-Zahlen sind und A wesentlich kleiner als B ist, ungleich A, denn bei der Addition können nicht alle Stellen von A berücksichtigt werden.

Die Variablen vom Typ `real` haben als Werte reelle Zahlen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Real-Konstanten

Für die Schreibweise von reellen konstanten Zahlen in Pascal gilt: Reelle Werte werden mit Dezimalpunkt oder mit Exponentialteil oder mit beidem geschrieben. Der Buchstabe E (oder e), der dem Exponenten vorangestellt ist, bedeutet "mal 10 hoch". Zu beachten ist, daß in einer reellen Zahl vor und nach dem Dezimalpunkt mindestens je eine Ziffer stehen muß.

Beispiele für vorzeichenlose reelle Zahlen:

0.6 5E-8 49.22E+08 1E10

Beispiele für falsch geschriebene Zahlen:

3,159 .6 E10 5.E-16 3.487.159

Ähnlich wie bei den ganzen Zahlen werden die Gleitkommazahlen nach ihrem Geltungsbereich und ihrer Genauigkeit unterteilt.

In Turbo Pascal gibt es außer `real` auch den Typ `double`, der eine höhere Genauigkeit und einen größeren Wertebereich hat. Die folgende Tabelle zeigt den Wertebereich und die Genauigkeit der beiden Typen in Turbo Pascal:

Typ	Wertebereich	Anzahl Dezimalstellen	Größe in Bytes
`real`	± 2.9e-39..1.7e38	11-12	6
`double`	± 5.0e-324 .. 1.7e308	15-16	8

Auch hier kann es zu Problemen kommen, wenn der zulässige Bereich überschritten wird.

5.2.3 Datentyp `boolean`

Ein **boolescher Wert** ist einer der logischen Wahrheitswerte, die durch die Standardbezeichner **`false`** und **`true`** repräsentiert werden.

Ein Variable des Typs `boolean` kann den Wert `false` oder `true` annehmen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

5.2.4 Datentyp `char`

Ein Wert vom Typ `char` ist ein Element einer endlichen geordneten Menge von Zeichen.

In einer endlichen geordneten Menge hat jedes Element bis auf das letzte einen Nachfolger und jedes Element bis auf das erste einen Vorgänger. Geordnet bedeutet, dass es eine Größer-Beziehung > zwischen zwei Elementen geben muss.

Es gibt keine allgemein anerkannte Standardzeichenmenge, so daß die Elemente und ihre Ordnung vom Zeichensatz des entsprechenden Rechners abhängen und durch sie definiert sind.

Der Datentyp `char` stellt also aus dem Zeichensatz des Rechners Buchstaben, Ziffern, Sonderzeichen und Steuerzeichen zur Verfügung. Zu den Sonderzeichen gehört auch das Leerzeichen (Blank).

Zeichenkonstanten

Konstanten dieses Typs werden durch ein Zeichen in einfachen Hochkomma (Apostroph) dargestellt. Um einen Apostroph selbst zu schreiben, muß man ihn doppelt schreiben. Es ist möglich, daß bestimmte Zeichenwerte nicht druckbar sind.

Beispiele:

```
'*'    'G'    '3'    ''''    ' " '
```

Unabhängig von der Implementierung gelten jedoch die folgenden minimalen Annahmen für den Typ `char`:

1. Die Dezimalziffern `'0'` bis `'9'` sind numerisch geordnet und aufeinanderfolgend.

2. Falls es Großbuchstaben `'A'` bis `'Z'` gibt, sind sie alphabetisch geordnet (z.B. `'A'<'B'`), folgen jedoch nicht notwendig unmittelbar aufeinander.[14]

3. Falls es Kleinbuchstaben `'a'` bis `'z'` gibt, sind sie alphabetisch geordnet (z.B. `'a'<'b'`), folgen jedoch nicht notwendig unmittelbar aufeinander.[15]

Die beiden Standardfunktionen `ord` und `chr` erlauben die Abbildung der Werte einer gegebenen Zeichenmenge auf ihre zugehörigen Ordinalzahlen (Ordinalzahlen: 0, 1, 2 ...) und umgekehrt; `ord` und `chr` werden **Umwandlungsfunktionen** genannt.

`ord(C)` ist die Ordinalzahl (die Nummer) des Zeichens `C` in der zugrundeliegenden ordinalen Zeichenmenge

`chr(N)` ist der Zeichenwert der Ordinalzahl `N`

[14] im ASCII-Zeichensatz folgen die Großbuchstaben unmittelbar aufeinander
[15] im ASCII-Zeichensatz folgen die Kleinbuchstaben unmittelbar aufeinander

Offensichtlich sind `ord` und `chr` zueinander invers, das heißt:

```
chr(ord(C)) = C
```

und

```
ord(chr(N)) = N
```

Die Ordnung einer gegebenen Zeichenmenge ist definiert durch:

`C1 < C2` genau dann, wenn `ord(C1) < ord (C2)`

Die Funktionen ord und chr werden in folgendem Beispiel nochmals verwendet.

Verschlüsselung von Zeichen

Bei einer einfachen Verschlüsselung, der Julius-Cäsar-Verschlüsselung werden die Buchstaben immer um eine konstante Anzahl Zeichen verschoben. Beispielsweise wird bei der Verschiebung um 1 Zeichen aus dem Zeichen `'A'` das Zeichern `'B'`, aus `'B'` wird `'C'` und so weiter. `'Z'` wird dann zu `'A'`.

Die folgenden Buchstabenreihen veranschaulichen diesen Zusammenhang. Die obere Reihe zeigt die Eingabezeichen, die untere die Ausgabezeichen.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
```

Das entsprechend Verschlüsselungsprogramm sieht so aus:

```
program chiffre1;

var zeichen : char;                {Vereinbarungsteil}

begin
  write ('Eingabezeichen :');
  readln (zeichen);
  if (zeichen < 'Z') then
    zeichen := chr(ord(zeichen) + 1) {hier kein Strichpunkt!}
  else
      zeichen := chr(ord(zeichen) - 25);
  writeln('Ausgabezeichen: ', zeichen);
end.
```

Programm 5-1 Verschlüsselungsprogramm mit chr und ord

Beachten Sie die Fallunterscheidung

```
if (zeichen < 'Z') then
    zeichen := chr(ord(zeichen) + 1)
  else
      zeichen := chr(ord(zeichen) -25)
```

Ist das Zeichen kleiner als `'Z'` so wird die Anweisung hinter `then` abgearbeitet, ansonsten die Anweisung hinter `else`. Die If-Anweisung wird ausfürlich in Kapitel 6.4.1 besprochen.

Mit der `ord()`-Funktion wird der ASCII-Wert eines Zeichens ermittelt. Die `chr()`-Funktion ermittellt dann wieder ein Zeichen aus dem ASCII-Wert.

5.2.5 Zeichenketten

Eine **Zeichenkette** oder ein **String** umfaßt eine Folge von Zeichen, die in einfache Hochkommas (Apostrophe) eingeschlossen sind. Soll eine Zeichenkette einen Apostroph enthalten, dann wird dieser doppelt geschrieben.

Zeichenketten, die aus nur einem Zeichen bestehen, sind die Werte, die die Variablen des Typs `char` annehmen können.

5.2.6 Datentyp `string` in Turbo Pascal

Eine **Variable vom Typ** `string` **(String-Variable)** wird vereinbart durch:

```
var variablenname : string [Zeichenzahl];
```

Die Zeichenzahl stellt dabei eine Integerzahl zwischen 1 und 255 dar.

Beispiel:

```
name : string [15];
```

Anmerkungen:

Der Compiler reserviert (Zeichenzahl + 1) Bytes für die Variable. In dem zusätzlichen Byte wird die **aktuelle Länge** des Strings eingetragen. Durch das zusätzliche Byte für die aktuelle Länge weiß die `write`-Anweisung von Turbo Pascal, wieviel Zeichen ausgegeben werden müssen. Die Zeichenzahl muß deshalb bei der `write`-Anweisung nicht explizit angegeben werden.

Wird bei der Vereinbarung `[Zeichenzahl]` weggelassen, so werden automatisch 255 Zeichen reserviert.

Die Länge einer Zeichenkette kann mit der Funktion `length(stringvariable)` abgefragt werden.

`Beispiel:`

```
var testwort : String;
..
testwort := 'Hallo';

writeln ( 'Länge von Hallo ', length(testwort));  {Ergebnis: 5}
```

5.2.7 Typanpassung

Einer Variablen dürfen nur Werte des gleichen Typs zugewiesen werden. Die Werte müssen von dem Typ sein, der für die Variable vereinbart wurde.

Es gibt eine Ausnahme: Man darf Variablen vom Typ `real` auch `integer`-Werte zuweisen. Dies kann damit begründet werden, daß ganze Zahlen eine Teilmenge der reellen Zahlen sind.

Wird einer `real`-Variablen ein `integer`-Wert zugewiesen, dann wird dieser in den entsprechenden `real`-Wert umgeformt und so abgespeichert. Man erhält ihn nicht als `integer`-Wert zurück.

Diese Umformung von `integer` nach `real` erfolgt **implizit**.

Eine **explizite** Typumwandlung von `real` nach `integer` kann mit Hilfe der Standardfunktionen `trunc` bzw. `round` erfolgen.

5.3 Aufgaben

Aufgabe 1: Schreibweise für Konstanten

Konstanten werden im Vereinbarungsteil vor den Variablen vereinbart wie in folgendem Besipiel zu sehen ist:

```
program mathe;

const PI = 3.141592
var x : real;
. . . . .
```

Prüfen Sie die Syntax der folgenden Konstanten mit Werten vom Typ `real` bzw. `integer` mit Hilfe eines Programms. Erklären Sie in Kommentaren, was die Fehlerquelle ist und wie die syntaktisch richtige Schreibweise aussieht.

a) const ALPHA = -1e-0;
b) const BETA = -e12
c) const GAMMA = .517
d) const DELTA = 3+

Aufgabe 2: Zahlenüberlauf

Unter dem Namen `maxint` steht die größte positive ganze Zahl eines Computers als Konstante zur Verfügung.

Interpretieren Sie sie Ausgaben des folgenden Programms:

```
program obergrenze (input, output);
var x: integer;
    y: integer;
begin
   writeln;
   writeln;
   writeln ('Hallo, hier ist Programm obergrenze');
   writeln ('maxint: ', maxint);
   x:= maxint;
   writeln ('x = maxint = ', x);
   y:= x;
   writeln ('y = maxint = ', y);
   y:= x - 1;
   writeln ('y = maxint - 1 = ', y);
   y:= x + 1;
   writeln ('y = maxint + 1 = ', y);
   y:= x + 2;
   writeln ('y = maxint + 2 = ', y);
   y:= x + 3;
   writeln ('y = maxint + 3 = ', y)
end.
```

Programm 5-2 Zahlenüberlauf bei Integer

Aufgabe 3: Unterstützte Datentypen bei read und write.

Zeigen Sie anhand eines Beispielprogramms, daß in Pascal mit der Standardfunktion `read` bzw. `readln` das Einlesen der Datentypen `char, integer` und `real` möglich ist, nicht aber das Einlesen des Datentyps `boolean`.

Zeigen Sie, daß mit Hilfe der Standardfunktion `write` das Ausgeben der Datentypen `char, integer, real, boolean`, sowie von Stringkonstanten (Zeichenketten) möglich ist.

(in Turbo Pascal ist auch `read` und `write` für `string`-Typen möglich - dies soll hier nicht gezeigt werden)

Aufgabe 4: Zeichen ein- und ausgeben und ihre Nummer im Zeichensatz bestimmen

Lesen Sie von der Tastatur Zeichen ein und geben Sie <CTRL> + <Z> ein, wenn Sie aufhören wollen. Beachten Sie, dass das <CTRL> + <Z> mit <CR> noch an das Programm übergeben werden muss. Jedes eingelesene Zeichen soll in einer eigenen Zeile mit `writeln` ausgegeben werden. In dieser Zeile ist außer dem Zeichen auch seine Nummer im Zeichensatz auszugeben.

```
.....
while (not eof)
   begin
      read (c);
      writeln (.....)
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
   end
{while (not eof) bedeutet: solange <CTRL> + <Z> noch nicht
eingegeben wurde}
```

Aufgabe 5: Zeichen zählen des Zeichenstroms von der Standarddatei input

Schreiben Sie ein Programm, das zählt, wieviel Zeichen vom Terminal eingelesen wurden. Die Zeichen sollen gezählt werden, bis eof true ist, d.h. bis Ctrl + Z eingegeben wird.

Aufgabe 6: Zeilen zählen des Zeichenstroms von der Standarddatei input

Schreiben Sie ein Programm, das die eingegebenen Zeilen zählt.

Aufgabe 7: Unterschied read/readln.

Testen Sie folgendes Programm mit verschiedenen Eingaben (ein, mehrere oder gar kein Zeichen eingeben.) Ersetzen Sie dann READ durch READLN und testen sie wieder. Versuchen Sie die Bildschirmausgabe zu erklären.

```
program lese;        {Abbruch des Programms durch Eingabe von #}
var  ch  :char;

begin
  write('ch: ');
  repeat
    read(ch);
    write(' ',Ord(ch));
  until ch = '#';
end.
```

Programm 5-3 Lesen mit read und readln

Aufgabe 8: Verschlüsselung von Zeichen

Schreiben Sie ein Programm, das Zeichen von der Tastatur einliest, bis <CTRL> + <Z> eingegeben wurde. Jedes eingelesene Zeichen soll verschlüsselt ausgegeben werden. Verwenden Sie den oben eingeführten Verschlüsselungsalgorithmus.

Aufgabe 9: Stringvariablen

Schreiben Sie ein Programm, das Ihren Namen und Ihren Vornamen jeweils in eine eigene Stringvariable einliest. Geben Sie den Namen und den Vornamen wieder aus.

Die Ausgabe sollte so ähnlich aussehen:

```
Bitte geben Sie Ihren Vornamen ein: Ariel  (Benutzereingabe)

Hallo Ariel!
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
Es freut mich, dass Sie sich mit Pascal beschäftigen.
```

Zusatz: Berechnen Sie zusätzlich die Länge des eingegebenen Namens und geben Sie diese aus.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

6 Lektion 6 - Anweisungen

Die wichtigsten Anweisungen sind

- die **Wertzuweisung**
- die **Verbundanweisung**
- die **bedingte Anweisung** (`if`, `case`)
- die **Wiederholungsanweisung** (`while`, `repeat`, `for`)

- der Aufruf einer **Prozedur**
- die **Sprunganweisung**
- die **leere Anweisung**

6.1 Wertzuweisung

Die Wertzuweisung hat die Aufgabe, den momentanen Wert einer Variablen durch einen neuen Wert zu ersetzen, der sich durch die Auswertung eines Ausdrucks ergibt.

Beispiele sind:

```
a := b + c;
j := b * (c + d * 123);
z := sin(x);
```

Ein Ausdruck ist eine Kombination von Operanden, Operatoren und runden Klammern.

Ein Operand ist beispielsweise eine Konstante oder eine Variable, ein Operator beispielsweise der Additions-Operator `+`.

Ganz wesentlich bei der Wertzuweisung ist, daß der Typ der Variablen mit dem Typ des Ausdrucks übereinstimmen muß.

Beispiele:

```
var      x    : real;
         y    : real;
         i    : integer;
         j    : integer;
         z    : char;
...

         y    :=  x + cos(x);
         i    :=  i + 1;
         { Es wird der Wert der Variablen i genommen, um 1
           erhöht, und der neue Wert wieder in der Variablen i
           abgelegt}
```

```
z    :=  'e';
```

Bei der verlangten Typengleichheit gibt es aus praktischen Gründen eine Ausnahme: hat der Ausdruck den Typ `integer`, so darf die Variable den Typ `real` haben. Dies gilt aber nicht umgekehrt!

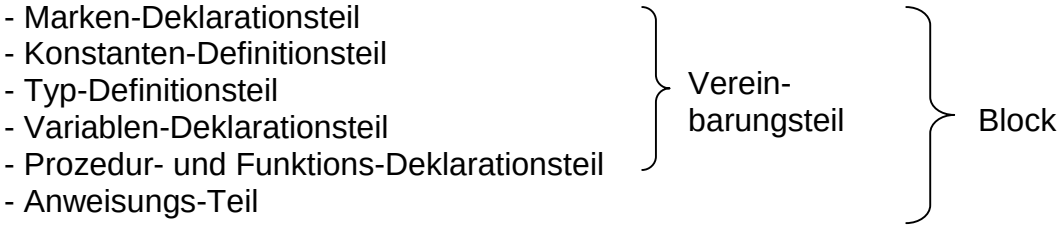
6.2 Verbundanweisung

Eine Verbundanweisung ist eine Anweisung, die in sich eine Folge einer oder mehrerer Anweisungen umfaßt. Diese Anweisungsfolge wird in der Reihenfolge ausgeführt, in der sie angeschrieben ist. Eingeschlossen wird die Anweisungsfolge durch die **Anweisungsklammern** `begin` und `end`.

Syntaktisch gilt sie als eine Anweisung. Will man an eine Stelle (z.B. `for`-Schleife, `if`-Anweisung etc.), an der nur eine Anweisung zugelassen ist, mehrere Anweisungen schreiben, so bedient man sich der Verbundanweisung.

Der Begriff **Verbundanweisung** ist identisch mit dem Begriff **Anweisungsteil**. Eine Verbundanweisung stellt einen **Block ohne Vereinbarungsteil** dar.

Ein Block besteht aus sechs Abschnitten, von denen außer dem letzten jeder leer sein kann und deren Reihenfolge durch die Blockdefinition festgelegt wird:



6.3 Wiederholungsanweisungen

Oftmals sollen gleichartige Handlungen wiederholt durchgeführt werden. Zum Beispiel soll für die Zahlen 1 bis 5 jeweils die Quadratzahl berechnet werden.

Ein möglicher Algorithmus zur Berechnung dieser Quadratzahlen könnte folgendermassen lauten:

Zahl ist gleich eins
solange die Zahl kleiner als 6 ist
    Bilde das Quadrat und erhöhe anschließend die Zahl um 1

Solange also das Abbruchkriterium 6 nicht erreicht ist, ist in einer Schleife die Berechnung der Quadratzahl durchzuführen und anschließend die Zahl um 1 zu erhöhen. Natürlich sollen die Zahlen und ihre zugehörigen Quadratzahlen auf dem Bildschirm ausgegeben werden.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Das zugehörige Programm lautet:

```
program quadratzahlen;

var lv           : integer;
    quadratzahl  : integer;

begin
writeln ('Zahl        Quadratzahl');
     lv := 1;

     while (lv < 6) do
         begin
           quadratzahl := lv * lv;
           writeln (lv:3, quadratzahl:16);
           lv := lv + 1; {Erhöhen des Zaehlers}
         end
end.
```

Programm 6-1 Quadratzahlen berechnen mit while

Beachten Sie, dass `lv` rechtsbündig mit einer Breite von 3 Stellen und die Quadratzahl ebenfalls rechtsbündig mit einer Breite von 16 Stellen ausgegeben wird. Dies wird durch `:3` bzw `:16` bewirkt. Bis zur vollen Stellenbreite wird nach links mit Leerschlägen aufgefüllt.

6.3.1 while-Anweisung

Die **`while`-Anweisung** ist eine **abweisende Schleife**, d.h. es wird erst geprüft, ob die entsprechende Bedingung[16] erfüllt ist, und erst dann wird die betreffende Anweisung abgearbeitet. Dabei wird die Bedingung zu Beginn eines jeden neuen Durchlaufs geprüft. Die Syntax der `while`-Anweisung ist:

```
WHILE (Bedingung) DO Anweisung
```

Ist zu Beginn der Schleife die `Bedingung true`, so muss sie während der Bearbeitung verändert werden, sonst entsteht eine Endlos-Schleife[17]. Die Schleife wird solange ausgeführt, wie der logische Ausdruck den Wert `true` hat. Es kann also sein, daß die Anweisung nach `do` überhaupt nicht ausgeführt wird.

Nach `do` darf **nur eine einzige Anweisung** stehen. Will man mehrere Anweisungen dort abarbeiten, muß man eine **Verbundanweisung** bilden.

Beispiel:

```
const  betrag = 287.5; {betrag ist eine Konstante}
```

[16] Eine **Bedingung** ist ein sogenannter **Boolescher Ausdruck.** Ein Boolescher Ausdruck[16] kann die Wahrheitswerte TRUE (wahr) bzw. FALSE (falsch = nicht wahr) annehmen.
[17] Eine **Endlos-Schleife** ist eine Schleife, deren Ausführung nie abbricht.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
var    summe  : real ;
       x      : real ;

...
   summe := 0;
   read(x);
   while (summe <= betrag) and not eof do
         begin
           summe := summe + x;
           write ('gib den nächsten Summanden ein: ');
           read (x);
         end
...
```

Hinweis: `eof` nimmt den Wert `true` an, wenn gleichzeitig die Tasten <CTRL> und <Z> gedrückt werden. <CTRL>[18] heisst bei manchen Tastaturen auch <STRG>[19].

Achtung: Wird der Ausdruck nicht `false`, so entsteht eine Endlosschleife.

6.3.2 repeat- Anweisung

Die **repeat-Anweisung** ist eine **annehmende Schleife**. Es wird auf jeden Fall ein Durchlauf gemacht und erst dann findet der Test auf ein Abbruchkriterium statt. Damit wird die `repeat`-Schleife mindestens einmal durchlaufen.

Die Syntax der repeat-Anweisung ist:

```
REPEAT Anweisungen UNTIL Bedingung
```

Die annehmende Schleife wird also mindestens einmal durchgeführt. Erst dann wird die Bedingung bewertet. Die `REPEAT`-Schleife wird typischerweise dann benutzt, wenn der Wert der Bedingung erst in der Schleife entsteht, beispielsweise wie in der folgenden Anwendung: „Lies Zahlen ein, bis eine 0 eingegeben wird“. Hier muss zuerst eine Zahl eingelesen werden. Erst dann kann geprüft werden, ob sie 0 ist oder nicht.

Beispiel:

```
var x     :  real;

begin

    ...    (* hier werden die Variablen initialisiert *);
    repeat
       writeln (‘Bitte Zahl eingeben:’));
       readln(x);
    until x = 0;
    ...
```

[18] CTRL heisst Control
[19] STRG heisst Steuerung

```
end.
```

Achtung: Wird der Ausdruck nicht `true`, so entsteht eine Endlosschleife.

Hier das Quadratzahlenprogramm mit der `repeat`-Anweisung:

```
program quadratzahlen2;

var lv           : integer;
    quadratzahl  : integer;

begin
   writeln ('Zahl        Quadratzahl');
   lv := 1;
   repeat
      begin
         quadratzahl := lv * lv;
         writeln (lv:3, quadratzahl:16);
         lv := lv + 1;
      end;
    until (n = 6);
end.
```

Programm 6-2 Quadratzahlen berechnen mit repeat

6.3.3 for-Anweisung

`for`-Schleifen werden typischerweise dann verwendet, wenn die Zahl der Durchläufe bereits vor dem Eintritt in die Schleife berechnet werden kann. Die `for`-Anweisung ist eine abweisende Schleife, d.h. es wird erst geprüft, ob die Bedingung erfüllt ist, und erst dann wird die betreffende Anweisung abgearbeitet.

Zu Beginn gleich zwei Beispiele:

Beispiel1:

```
var i:   integer

for i := 2 to 5 do write(i);
```

Dieses Beispiel gibt die Zahlen 2, 3, 4 und 5 aus.

Beispiel2:

```
for i := 7 downto 4 do write(i);
```

gibt die Zahlen 7, 6, 5 und 4 aus.

Ist die Laufvariable vom Typ `integer`, so ist die Schrittweite bei `to` immer 1, bei `downto` immer -1.

Die Variable nach `for` ist die sogenannte **Laufvariable**. Die Zuweisung an die Laufvariable liefert den **Anfangswert**, der Wert hinter `to` bzw. `downto` den **Endwert** der Laufvariablen. Laufvariable und die Ausdrücke für den Anfangs- und den Endwert müssen vom selben Typ sein, der ein Ordinaltyp[20] sein muß. In diesem Kurs beschränken wir uns bei Laufvariablen auf den Ordinaltyp `integer`.

Generell ist zu beachten:

- Laufvariable, Anfangswert und Endwert dürfen in der Anweisung nach `do` nicht verändert werden.
- Ist eine `for`-Anweisung ausgeführt worden, so ist danach der Wert der Laufvariablen nicht definiert. Wird eine `for`-Anweisung hingegen durch eine Sprunganweisung[21] verlassen, so hat die Laufvariable danach den Wert, den sie im Moment des Verlassens hatte.
- Ist der Startwert gleich dem Endwert, wird die Schleife einmal durchlaufen

Hier das Quadratzahlenprogramm mit der `for`-Anweisung:

```
program quadratzahlen3;

var lv           : integer;
    quadratzahl  : integer;

begin
   writeln ('Zahl        Quadratzahl');
   for lv := 1 to 5 do
      begin
         quadratzahl := lv * lv;
         writeln (lv:3, quadratzahl:16);
      end;
end.
```

Programm 6-3 Quadratzahlen berechnen mir for

6.4 Fallunterscheidungen

Ein Ausdruck vom Typ Boolean kann bekanntermaßen zwei Werte annehmen, `true` oder `false`. Die if-Anweisung bietet die Möglichkeit, je nachdem welcher Wert der boolesche Ausdruck zur Laufzeit annimmt, die hierfür vorgesehenen Anweisungen auszuführen.

Je nachdem, ob der Audruck `true` oder `false` ist, kann das Programm etwas anderes tun.

6.4.1 if-Anweisung

[20] Ein Ordinaltyp hat diskrete Werte, die geordnet sind
[21] Sprunganweisungen werden in diesem Kurs nicht behandelt

Einfache Alternative

Die Syntax der `if`-Anweisung ist gegeben durch:

```
IF (Bedingung) THEN Anweisung1
ELSE Anweisung2
```

Ist der logische Ausdruck nach der `if`-Anweisung `true`, wird die Anweisung nach `then` ausgeführt, ansonsten die Anweisung nach `else`. Dies ist die sogenannte **einfache Alternative.**

Beispiel:

```
if (a = 0) writeln ('a ist Null')
else writeln ('a ist von Null verschieden')
```

Hinweis: Vor `else` darf niemals ein Semikolon stehen

Bedingte Anweisung

Ist im Falle `false` nichts zu machen, kann der Teil mit `else` auch entfallen. Dies ist die sogenannte **bedingte Anweisung**. Hier wird also der `true`-Zweig angesprungen, wenn der Ausdruck wahr ist. Ansonsten wird direkt die nach der bedingten Anweisung folgende Anweisung aufgerufen.

Beispiel:

```
if (fehlerfall = true) fehlerroutine (fehlernummer)
```

Im Fehlerfall wird eine Fehlerroutine aufgerufen, die den Fehlertext ausgibt und das Programm beendet.

Nach `then` und `else` ist jeweils nur 1 Anweisung erlaubt. Sind nach `then` oder `else` mehrere Anweisungen notwendig, so sind diese wieder als Verbundanweisung zu schreiben.

Schachtelung von if-Anweisungen

Die Schachtelung von if-Anweisungen kann problematisch sein, da man der geschachtelten Anweisung

```
if ausdruck_1 then if ausdruck_2 then anweisung_1 else anweisung_2
```

nicht ohne weiteres ansieht, zu welchem `if` das `else` gehören soll.

Pascal wertet diesen Ausdruck folgendermaßen aus:

```
if ausdruck_1 then
   begin
      if ausdruck_2 then
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
         anweisung_1
      else
         anweisung_2
   end
```

Sicherheitshalber sollte man also mit Hilfe von Verbundanweisungen arbeiten oder auf geschachelte `if`-Anweisungen ganz verzichten, es sei denn, man verwendet die **mehrfache Alternative** oder die **mehrfache Alternative mit else-Teil.**

Mehrfache Alternative:

```
if (ausdruck = alternative_1) then anweisung_1
else if (ausdruck = alternative_2) then anweisung_2
else if (ausdruck = alternative_3) then anweisung_3
                  .
                  .
                  .
else if (ausdruck = alternative_n) then anweisung_n
```

Bei der mehrfachen Alternative wird geprüft, ob der Ausdruck einen von n vorgegebenen Werten annimmt. Ist dies der Fall, so wird der entsprechende Zweig angesprungen, ansonsten wird direkt zu der nächsten Anweisung übergegangen.

Mehrfache Alternative mit else-Teil:

```
if (ausdruck = alternative_1) then anweisung_1
else if (ausdruck = alternative_2) then anweisung_2
else if (ausdruck = alternative_3) then anweisung_3
                   .
                   .
                   .
else if (ausdruck = alternative_n) then anweisung_n
else ausdruck_fuer_else_teil
```

Bei der mehrfachen Anweisung mit `else`-Zweig wird der `else`-Zweig angesprungen, wenn der Ausdruck mit keinem der vorgegebenen Werte übereinstimmt.

Beispiel für eine Mehrfachverzweigung:

```
if error_code = 0 then
    writeln ('Warnung')
else if error_code < 0 then
    writeln ('Fataler Fehler')
else
    writeln ('Aufruf erfolgreich');
```

6.4.2 case-Anweisung

Eine Alternative zur mehrfachen Alternative mit `if` stellt die `case`-Anweiung dar. Aus Zeitgründen wird sie aber bei diesem Kurs nicht besprochen. Mehr Informationen bekommt man in der Online-Hilfe von Turbo Pascal mit STRG + F1.

6.5 Sprunganweisung

Entfällt bei diesem Kurs

6.6 Leere Anweisung

Die Einführung der leeren Anweisung vereinfacht den Umgang mit Verbundanweisungen. Eine Verbundanweisung hat die Form

```
begin
   anweisung;
   anweisung;
     ...
   anweisung
end
```

Würde man jetzt die letzte Anweisung vor dem **end** löschen oder eine Anweisung dort einfügen, so müßte man immer sehr sorgfältig das Setzen des Strichpunktes entsprechend nachziehen.

Mit dem Einführen der leeren Anweisung ist auch

```
begin
   anweisung;
   anweisung;
     ...
   anweisung;
end
```

korrekt.

Der Compiler interpretiert die Folge `;end` in der Weise, daß er zwischen **;** und **end** eine leere Anweisung annimmt, womit der Strichpunkt seine Rolle als Trennzeichen zwischen Anweisungen beibehält.

6.7 Aufgaben

Aufgabe 1: Schleifen.

Es werden interaktiv die Integer-Zahlen A und B eingegeben und aufaddiert, solange bis A = 0 eingegeben wird. Die Summe aller Zahlen ist auszugeben.

Hinweis: Verwenden Sie die **repeat-Schleife**

Aufgabe 2: Fallunterscheidung

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Schreiben Sie ein Programm, mit dem zwei ganze Zahlen dividiert werden. Das Ergebnis soll vom Typ real sein. Geben Sie eine Fehlermeldung aus, wenn der Divident 0 ist.

Aufgabe 3:Verschachtelte Verzweigungen

Die Zeichen auf der Tastatur kann man in Buchstaben, Ziffern und sonstige Zeichen einteilen. Schreiben Sie ein Programm, das nach der Eingabe eines Zeichens ermittelt, ob es sich um einen Buchstaben, eine Ziffer oder um ein sonstiges Zeichen handelt. Dazu soll der Computer einen entsprechenden Text ausgeben.

Zeichen	ASCII-Bereich
Großbuchstaben	65 - 90
Kleinbuchstaben	97 - 122
Ziffern	48 - 57
sonst	Rest

Zusätzlich sollen die Vokale A,E,I,U erkannt werden. Verschachteln Sie dazu die obige Anweisung entsprechen.

Hinweis: logische Ausdrücke wurden noch nicht behandelt. Benutzen Sie für die Vokale die folgende if-Anweisung als Vorlage.

```
If ((buchstabe = 'A') or (buchstabe = 'E') or (buchstabe = 'I')
or ...)
```

Aufgabe 4: Sortieren

Es werden interaktiv die reellen Zahlen A, B und C eingegeben. Die kleinste Zahl, die größte Zahl und der Mittelwert werden bestimmt und am Bildschirm ausgegeben.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

7 Lektion 7 - Elemente von Pascal

Nachdem nun schon einige Erfahrungen im Programmieren gesammelt wurden, sollen im folgenden die in Pascal zulässigen Zeichen und die in Pascal generell möglichen Wortarten (lexikalischen Einheiten, Symbole) betrachtet werden.

7.1 Zeichenvorrat

[Quellen: R. Herschel: Standard-Pascal, K. Jensen, N. Wirth: Pascal Benutzerhandbuch]

Ein Pascal-Programm wird geschrieben als eine Folge von Zeilen, wobei der Text des Programms Zeichen aus dem folgenden **Zeichenvorrat** (Zeichensatz, character set) enthalten kann.

Buchstaben:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Ziffern:

0 1 2 3 4 5 6 7 8 9

Sonderzeichen:

+ - * / = < > ( ) [ ] { } : . , ' ; _ ↑ ⊥

Das Zeichen ⊥ soll an dieser Stelle das Zeichen für Blank oder Zwischenraum versinnbildlichen. Ein Blank oder Zwischenraum wird erzeugt durch einen Leerschlag (dieser wäre im Zeichensatz oben jedoch nicht sichtbar, daher die besondere Kennzeichnung an dieser Stelle).

Dieser Zeichenvorat kommt auf jeder **üblichen** Tastatur vor, allerdings sind zum Generieren einzelner dieser Zeichen auf manchen Tastaturen spezielle Tastenkombinationen erforderlich.

Kleinbuchstaben und Großbuchstaben werden grundsätzlich als gleichwertig angesehen. Die einzige Ausnahme sind Buchstaben, die in **Zeichenketten** oder für sich allein in **Zeichenliteralen** (Zeichenketten aus einem Zeichen) stehen.

So sind beispielsweise:

`'A'` verschieden von `'a'` (`'A'` und `'a'` sind Zeichenliterale)

`'Hallo'` verschieden von `'hallo'` (`'Hallo'` und `'hallo'` sind Zeichenketten)

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

7.2 Symbole und Trenner

Pascal-Programme bestehen aus Symbolen und Symboltrennern.

Ein Symbol (eine lexikalische Einheit) ist nichts anderes als ein Wort der Sprache.

Ein Symboltrenner trennt zwei Wörter. Der einfachste Symboltrenner ist ein Leerzeichen.

Symboltrenner (Zwischenräume und Kommentare) dienen dem Compiler dazu, zu erkennen, wo ein Wort zu Ende ist. Ansonsten haben sie keine Bedeutung für den Compiler und werden überlesen.

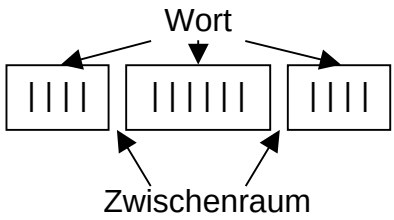


Bild 7-1 Erkennen von WörternSymbole

Man kann sich eine Zeile des Pascal-Textes als eine Folge von Zeichengruppen vorstellen, die lexikalische Einheiten oder **Symbole** genannt werden.
(Anmerkung: lexikalisch im sprach-wissenschaftlichen Sinne bedeutet: ein Wort - ohne Textzusammenhang - betreffend).

Die Symbole von Pascal umfassen

- die Spezialsymbole
- die Wortsymbole
- die Bezeichner (Namen)
- die Zahlen
- die Zeichenketten[22]

7.2.1 Trenner

Zwischenräume (Leerzeichen, Blank), Zeilenenden (Zeilentrennungen) oder **Kommentare** werden als **Trenner für Symbole** angesehen.

Kein Teil eines Trenners darf in einem Pascalsymbol auftreten. Zwischen zwei aufeinanderfolgenden Bezeichnern, Wortsymbolen oder Zahlen muß mindestens ein Trenner eingefügt sein.

[22] - die Marken und Direktiven werden hier nicht behandelt.

Kommentare

Kommentare dienen dazu, die Bedeutung von Anweisungen und Programmeinheiten schriftlich direkt an der entsprechenden Stelle im Quellcode[23] festzuhalten. Kommentare werden vom Compiler ignoriert und dienen nur zur Dokumentation des Quellcodes. Sie sind oftmals genau so wichtig wie das eigentliche Programm, da anhand von ihnen das Programm besser verstanden werden kann, als wenn die Funktion nur durch Analyse der Anweisungen erfolgt.

Generell lässt sich sagen, dass bei guten Programmen mindestens 10 - 20% des Quellcodes aus Kommentaren bestehen sollten. Es hat sich als guter Stil eingebürgert, sowohl das Programm als gesamtes als auch einzelne Unterprogramme und kleinere Abschnitte zu kommentieren. Oft enthält der Programmkommentar Informationen über das Programm, das Erstellungsdatum, den Autor, Änderungen des Programms mit laufender Nummer und Namen des Ändernden oder Erweiterungsmöglichkeiten.

In Pascal wird der Kommentar zwischen zwei geschweiften Klammern geschrieben. Ein Kommentar kann auch über mehrere Zeilen gehen. Als Beispiel das Begruessung-Programm mit Kommentaren.
/

```
{  Dieser Kommentar beschreibt das Programm begruessung

   Autor: Johannes-Markus Waizenegger (jmw)
   Firma: Steinbeis-Transferzentrum Softwaretechnik
   Entstehungsdatum:  20. 07. 2000

   Änderungen:
   1.  21.07.00   jmw   Programmfunktionen kommentiert
   2   24.07.00   jmw   Ausgabetext geändert
   3   30.08.00   jmw   Programmname geändert
}

program begruessung;

begin
  write ('Hallo, ich begruesse Sie zum Pascal-Kurs.'); {Ausgabe}
end.
```

Programm 7-1 Das begruessung-Programm mit Kommentaren

Ein **Kommentar** beginnt mit { oder (* (nicht innerhalb eines Kommentars) und endet mit } oder *). Ein Kommentar kann eine beliebige Folge von Zeilenenden und Zeichen, außer } oder *) enthalten. Dies hat die Konsequenz, daß geschachtelte Kommentare in Standard-Pascal nicht möglich sind.

[23] Quellcode ist das vom Programmierer in einer Programmiersprache geschriebene Programm.

Im Gegensatz zu Standard-Pascal ist in Turbo-Pascal eine Schachtelung von Kommentaren möglich. Dabei ist bei einer Schachtelung jeweils das andere Begrenzerpaar zu verwenden, beispielsweise also

{ (* { } *) }

Von praktischer Bedeutung ist eine Kommentarschachtelung beim Test eines Programms, wenn man beispielsweise ein Stück Programmtext, das selbst Kommentare enthält, mit Hilfe von Kommentarklammern ausblenden möchte.

Für das praktische Arbeiten mit Turbo Pascal wird empfohlen, im Programmtext stets nur eine der beiden Schreibweisen (* , *) oder {, } zu benutzen, keinesfalls aber eine gemischte Schreibweise zu verwenden. Damit kann man bei Bedarf mit Hilfe der jeweils anderen Schreibweise ohne Aufwand geschachtelte Kommentare erzeugen.

Oft läßt sich durch das Einfügen von Leerzeichen, Zeilenenden (leeren Zeilen) und Kommentaren die **Lesbarkeit** eines Pascal-Programms verbessern.

Die Beschreibung eines Programms verlangt darüber hinaus aber noch eine Reihe von Spezialsymbolen und von Wortsymbolen.

7.2.2 Spezialsymbole

Spezialsymbole können aus einem oder 2 Zeichen bestehen. Spezialsymbole, die aus zwei Zeichen bestehen, werden zusammenhängend geschrieben, ohne einen Trenner dazwischen. D.h., es werden zwei Sonderzeichen zusammengefaßt, die zusammen eine bestimmte Bedeutung haben. So wird etwa <= für "kleiner gleich" genommen oder <> bedeutet "ungleich". Die beiden Zeichen müssen, wie schon erwähnt, unmittelbar hintereinander geschrieben werden.

Spezialsymbole sind:

```
+ - * /
. , : ;
= <> < <= >=
:= .. _
( ) [ ]
```

7.2.3 Wortsymbole

Für die Steuerung des Programmablaufs ist eine große Zahl von Symbolen notwendig, für die man zweckmäßigerweise Wörter nimmt. In Pascal sind dies die **Wortsymbole**

(**reservierte Worte, Schlüsselwörter**)[24]. Diese wollen wir nicht im Detail betrachten. Der Vollständigkeit sind sie jedoch zum Nachschlagen aufgeführt:

```
and
array
begin
case
const
div
do
downto
else
end
file
for
forward
function
goto
if
in
label
mod
nil
not
of
or
packed
procedure
program
record
repeat
set
then
to
type
until
var
while
with
```

Wortsymbole dürfen nicht in einem anderen als dem durch die Pascal-Definition gegebenen Zusammenhang verwendet werden, insbesondere dürfen diese Wörter nicht als Bezeichner verwendet werden.

Ersatzdarstellungen für Spezialsymbole:

(. für [
.) für]
@ oder ^ für ↑

[24] In diesem Kurs werden nicht alle Wortsymbole behandelt

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Weitere Ersatzdarstellungen für die deutsche Tastatur[25]:

(* für {
und *) für }

7.2.4 Namen

In einem Programm braucht man zur Bezeichnung gewisser Größen **Namen** (häufig auch **Bezeichner** genannt). In Pascal werden

- **Konstanten**
- **Datentypen**
- Grenzen (siehe "Konformreihungsparameter" in der Literatur, hier nicht behandelt)
- **Variablen**
- **Programme**
- **Prozeduren**
- **Funktionen**

mit Namen bezeichnet.

Ein Name ist eine beliebig lange Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnen muß Bei manchen Pascal-Versionen gehört auch der Unterstrich _ zu den Buchstaben, so dass man z.B.

```
x_Quadrat
```

als Namen verwenden kann.

Während der Benutzer beliebig lange Namen wählen kann, bricht der Compiler den Namen nach einer bestimmten Anzahl von Zeichen ab. Der Benutzer muß dann darauf achten, daß sich die verschiedenen Namen in den für den Compiler relevanten Stellen unterscheiden. Wegen der Lesbarkeit des Programms sollte der Nutzer möglichst aussagekräftige Namen verwenden.

Namen dürfen erst verwendet werden, wenn ihre Deklaration im Text vorausgegangen ist.

Es gibt **Standardnamen (Standardbezeichner)**, die ohne weitere Definition in einem ganz bestimmten vordefinierten Sinne verwendet werden können. Dazu zählen Funktionen wie

```
sin
cos
ln
```

die Prozeduren

```
read
write
```

[25] Die meisten deutschen Tastaturen können mittlerweile die Zeichen { und } schreiben. Diese Ersatzdarstellungen sind historisch bedingt.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

für die Ein- und Ausgabe,

```
real
integer
```

als Namen für die Datentypen, etc.

8 Lektion 8 - Ausdrücke

Ein Ausdruck ist ein Gebilde aus Operanden, Operatoren und runden Klammern.

Ein Ausdruck ist eine **Berechnungsregel**, die nach ihrer Ausführung einen **Wert** liefert[26]. Der abgelieferte Wert hängt von den Werten der Konstanten und Variablen im Ausdruck ab, den Funktionen, die der Ausdruck aufruft und natürlich von den Operatoren. Steht eine Variable in einem Ausdruck, so ist ihr Wert zu nehmen, steht eine Funktion in einem Ausdruck, so ist ihr Rückgabewert, d.h. das Ergebnis der Funktion, zu nehmen.

Ein Ausdruck steht stellvertretend für den Wert, der sich ergibt, wenn man den Ausdruck auswertet, d.h. die Operanden entsprechend den Operatoren und Klammern miteinander verknüpft.

Da in Pascal ein Wert mit einer Typangabe verbunden ist, ist es von großer Bedeutung, genau zu wissen, von welchem Typ der Wert eines Ausdrucks ist.

Im folgenden wird beschrieben, wie Ausdrücke gebildet werden können, die einen Wert vom Typ integer oder real haben könen (die sogenannten **arithmetischen Ausdrücke**) oder einen Wert vom Typ boolean (die sog. **logischen Ausdrücke**).

8.1 Prioritäten der Operatoren

Bei der Auswertung eines Ausdrucks wird nach den folgenden vier Regeln verfahren:

1. Die Operatoren werden in vier Stufen eingeteilt:

   0 `not`
   1 Multiplikation: `* / div mod and`
   2 Addition: `+ - or`
   3 Vergleich: `= < > <= >= <> in`[27]

2. Die Operatoren einer niederen Stufe werden vor den Operatoren einer höheren Stufe ausgeführt.

3. Die Operatoren gleicher Stufe werden von links nach rechts ausgeführt

4. Das Innere von runden Klammern wird vor dem Äußeren ausgeführt. Damit kann die Regel 2 durchbrochen werden, wenn Operatoren einer höheren Stufe vor denen einer niederen ausgeführt werden sollen.

[26] außer der Ausdruck ruft eine Funktion auf, deren Abarbeitung durch eine GOTO-Anweisung beendet wird (siehe hierzu Jensen, Wirth: Pascal Benutzer-handbuch, Sprachbericht, Abschnitt 8, 9.1.3 und 10).

[27] Der Operator in betrifft Mengen und wird hier nicht behandelt

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

8.2 Arithmetische Ausdrücke

Arithmetische Ausdrücke haben einen Wert vom Typ `integer` oder `real`. Es müssen alle Operanden von einem dieser Typen sein. Als Operatoren gibt es:

Stufe	Operator	Operandentyp	Ergebnistyp	Bedeutung
1	*	real	real	Multiplikation
	*	integer	integer	Multiplikation
	*	integer,real	real	Multiplikation
	/	integer	real	Division
	/	real	real	Division
	/	integer,real	real	Division
	div	integer	integer	Division ohne Rest
	mod	integer	integer	modulus (Rest der Division)
2	+	integer	integer	Addition
	+	real	real	Addition
	+	integer,real	real	Addition
	-	integer	integer	Subtraktion
	-	real	real	Subtraktion
	-	integer,real	real	Subtraktion

Bild 8-1Arithmetische Operationen

Es gibt zwei Arten der Division. Die Division mit `/` führt immer zum Typ `real`, auch wenn Zähler und Nenner beide vom Typ `integer` sind. Beispiel:

7/3 = 2.3333333

Die Division mit `div` bedeutet die ganzzahlige Division unter Vernachlässigung des Restes. Beispiel:

7 div 3 = 2

Den Rest einer ganzzahligen Division erhält man mit dem Operator `mod`. Beispiel:

7 mod 3 = 1

In Ausdrücken können Funktionen aufgerufen werden. In arithmetischen Ausdrücken können Funktionen mit einem Typ `integer` oder `real` aufgerufen werden. (Anmerkung: In logischen Ausdrücken Funktionen vom Typ `boolean`).

In Pascal gibt es eine Reihe von Standardfunktionen, die ohne besondere Definition aufgerufen werden können (siehe Bild 8-2)

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Funktion	Argumenttyp	Ergebnistyp	Bedeutung
`abs(x)`	`integer`	`integer`	Betrag von x
`abs(x)`	`real`	`real`	Betrag von x
`arctan(x)`	`real,integer`	`real`	Hauptwert von arctan im Bogenmaß
`cos(x)`	`real,integer`	`real`	cos im Bogenmaß
`exp(x)`	`real,integer`	`real`	Exponentialfunktion
`ln(x)`	`real,integer`	`real`	natürlicher log
`sin(x)`	`real,integer`	`real`	sin im Bogenmaß
`sqr(x)`	`integer`	`integer`	Quadrat von x
`sqr(x)`	`real`	`real`	Quadrat von x
`sqrt(x)`	`integer,real`	`real`	Quadratwurzel von x (x>=0)
`ord(x)`	Ordinaltyp	`integer`	Nummer von x in dem Ordinaltyp
`trunc(x)`	`real`	`integer`	größte ganze Zahl <= x (x>=0)
			kleinste ganze Zahl >= x (x<0)
`round(x)`	`real`	`integer`	`round(x) = trunc(x+0.5)` (x>=0)
			`round(x) = trunc(x-0.5)` (x<0)

Bild 8-2 Arithmetische Standardfunktionen

Argument einer solchen Funktion kann nicht nur eine Variable, sondern auch ein Ausdruck von dem angegebenen Typ sein. Beispiele für die Übergabe von Ausdrücken sind:

```
sin(x+y), sqrt(1 + x*x), ln(a + b/2)
```

Die Funktionen `trunc` und `round` dienen der gezielten Umwandlung einer `real`-Zahl in eine `integer`-Zahl. Bei der Funktion `trunc` wird einfach der gebrochene Teil weggelassen:

```
trunc(2.7)     = 2
trunc(-4.6)    = -4
```

Bei der Funktion `round` wird im mathematischen Sinne gerundet:

```
round(2.7)     = 3
round (-4.6)   = -5
```

8.3 Logische Ausdrücke

Logische Ausdrücke haben einen Wert vom Typ `boolean`.

Ein Ergebnis vom Typ `boolean` entsteht beispielsweise bei der Anwendung von **relationalen Operatoren** (**Vergleichsoperatoren** wie z.B. >=, <> etc.) auf Operanden (Ausdrücke) vom einfachen Typ.

Relationale Operatoren dienen dazu, zwei Ausdrücke auf eine bestimmte Eigenschaft hin miteinander zu vergleichen und in Abhängigkeit davon, ob diese Eigenschaft besteht, einen **Wahrheitswert** (**boolesche Konstante**) zurückzugeben. Dabei müssen die beiden miteinander zu vergleichenden Operanden beide vom selben Typ sein mit der Ausnahme, daß man auch einen **real**-Wert mit einem **integer**-Wert vergleichen kann.

Die beiden Operanden eines Vergleichs können natürlich auch durch einen Ausdruck gegeben sein, wie z.B.:

```
x + y < 12.5
x := a + b >= c + d
```

Vergleiche dienen oft dazu, Iterationen (Schleifen) und Selektionen (`if`-Anweisung) zu steuern.

Die folgende Tabelle gibt einen Überblick über Operationen, die auf logische Ausdrücke führen:

Stufe	Operator	Operandentyp	Bedeutung
0	`not`	`boolean`	Negation
1	`and`	`boolean`	UND
2	`or`	`boolean`	ODER
3	= <> < <= > >=	einfacher Typ einfacher Typ einfacher Typ einfacher Typ einfacher Typ einfacher Typ	gleich ungleich kleiner kleiner oder gleich größer größer oder gleich

Bild 8-3 Logischen Operatoren und ihre Priorität

Wie bereits in Kapitel 8.1 besprochen, wird mit der kleinsten Stufe begonnen. Bei logischen Operatoren hat der **not**-Operator die höchste Priorität. Die nächste Priorität hat dann **and**, dann **or** und zuletzt die Vergleiche.

Wie bei den arithmetischen Audrücken werden auch logische Ausdrücke bei gleicher Priorität von links nach rechts abgearbeitet.

Auch logische Audrücke kann man klammern. Der Inhalt der Klammer wird dann vor dem restlichen Ausdruck berechnet.

So ergiebt

```
not true or true                    true      und
not (true or true)                  false
```

Will man bei einer Bedingung zwei Vergleiche anstellen, so muss man die Vergleiche klammern. Mit dem folgenden Ausdruck wird geprüft, ob eine Variable x zwischen 10 und 74 ist:

```
(x > 10) and (x < 74)
```

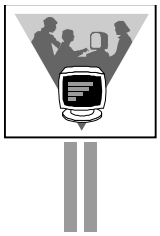
Logische Operatoren not, and, or

Im folgenden seien p und q boolesche Variablen:

```
var p,q    : boolean;
```

Das nachfolgende Programm soll die einzelnen logischen Operatoren und ihre Verwendung vorstellen.

```
program bool1;

var a, b, c, d : boolean;
var zahl1, zahl2; integer;

begin
  zahl1 := 1;
  zahl2 := 2;

  a := zahl1 < zahl2;
  writeln(a);

  b := not a; {logischer Operator not}
  writeln(b);

  c := a and b; {logischer Operator and}
  writeln(c);

  d := a or b;  {logischer Operator or}
  writeln(d);
end.
```



Hier die Programmausgabe:

```
true
false
false
true
```

Logisches NICHT

Beispiel:

```
var p,z : boolean;

z := not p
```

Der einstellige Operator `not` dient dazu, den Wahrheitswert eines logischen Ausdrucks zu negieren. Das Ergebnis der Operation ist also `false`, wenn der Operand `true` war und umgekehrt.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

p	z	
`true`	`false`	aus `true` wird `false`
`false`	`true`	aus `false` wird `true`

Bild 8-4Wahrheitstabelle für `not`

Logisches UND

Beispiel:

```
var p,q,z : boolean;
z := p and q
```

Der logische Operator `and` liefert genau dann den Wahrheitswert `true`, wenn beide Operanden den Wahrheitswert `true` haben. Hat hingegen mindestens einer der Operanden den logischen Wert `false`, so liefert die UND-Operation den Wert `false`.

p	q	z
false	false	false
false	true	false
true	false	false
true	true	true

Bild 8-5 Wahrheitstabelle für and

Logisches ODER

Beispiel:

```
var p,q,z : boolean;
z:= p or q
```

Der Operator `or` liefert genau dann den Wahrheitswert `true`, wenn mindestens einer der beiden Operanden den Wahrheitswert `true` hat. Haben beide Operanden den Wahrheitswert `false`, so ist das Ergebnis `false`.

p	q	z
false	false	false
false	true	true
true	false	true
true	true	true

Bild 8-6 Wahrheitstabelle für or

Logische Standardfunktionen

Die folgenden booleschen Standardfunktionen werden in Pascal bereit gestellt:

`odd(G)` liefert `true`, falls die ganze Zahl G ungerade ist.

`eoln(F)` ist `true`, wenn das Zeilenende der aktuellen Zeile in der Textdatei `F` erreicht ist (siehe später).

`eoln` ist `true`, wenn das Zeilenende der aktuellen Zeile von der Standarddatei `input` erreicht ist. Hierbei ist das Einlesen von der Standarddatei `input` in der Regel durch das Lesen des Zeichenstroms von der Tastatur realisiert.

`eof(F)` ist `true`, wenn das Ende der Datei `F` erreicht ist (siehe später).

`eof` ist `true`, wenn das Eingabe-Ende der Standarddatei `input` erreicht ist. Hierbei ist das Einlesen von der Standarddatei `input` in der Regel durch das Lesen des Zeichenstroms von der Tastatur realisiert. `eof` wird in Turbo Pascal dann generiert, wenn der Benutzer <CTRL> + <Z> eingibt.

8.4 Aufgaben

Aufgabe 1: Programm zur Temperaturumwandlung. Standardprozedur write. While-Anweisung. Arithmetische Ausdrücke

Unter Verwendung der Formel

Grad Celsius = (5/9) * (Grad Fahrenheit - 32)

soll eine Temperaturtabelle in Fahrenheit und Celsius in folgender Form auf dem Bildschirm ausgegeben werden:

Fahrenheit ***Celsius***

```
  0          -17
 20           -6
 40            4
 60           15
 ...
300          148
```

Hinweis: Verwenden Sie hierzu die **while**-Schleife

a) Führen Sie Berechnung mit `integer`-Variablen durch.
b) Führen Sie die Berechnung mit `real`-Variablen durch.

Aufgabe 2: Die Standardfunktionen trunc, round, odd, div, mod

Welchen Wert ergibt

a) trunc(-11.7) b) round(11.7) c) round(-11.5)
d) odd(11) e) 19 div 82 f) 19 mod 82

Schreiben Sie hierzu ein Programm und schreiben Sie die Ergebnisse als Kommentar in das Programm.

Aufgabe 3: Logische Operatoren

Gegeben sei der folgende Programmausschnitt:

```
VAR    A       :    boolean;
       B       :    boolean;
       C       :    boolean;
       D       :    boolean;
       E       :    boolean;
...
C:= (A and not B);
D:= (not A and B);
E:= C or D;
```

Gesucht: Werte von C, D, E für die Fälle

a) A = true, B = true b) A = true, B = false

Erstellen Sie eine Wahrheitstabelle durch Überlegen in der folgenden Form:

A	not A	B	not B	C	D	E
true	...	**true**	...	...	...	
true	...	**false**	...	...	...	

Prüfen Sie die Ergebnisse mit Hilfe eines Programmes.
Schreiben Sie die Ergebnisse als Kommentar in das Programm.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

9 Lektion 9 - Ein- und Ausgabe

Für die Ein- und Ausgabe gibt es in Pascal die beiden Standardprozeduren

`read (f)` und `write (f)`.

`read` und `write` lesen bzw. schreiben von/auf Dateien in einem verallgemeinerten Sinne. `f` ist hierbei der Name der Datei, wie sie im Programm angesprochen wird. Der Name der Datei im Dateisystem des Betriebssystems kann ein anderer sein. Die Zuordnung zwischen beiden Namen erfolgt in Turbo Pascal mit der Programm-Anweisung `assign`. Hierauf soll an dieser Stelle jedoch nicht näher eingegangen werden.

Worauf es hier jedoch ankommt, ist, zu wissen, daß die Ausgabe auf den Bildschirm behandelt wird als Schreiben in eine Datei, und daß das Lesen von der Tastatur behandelt wird als das Lesen von einer Datei.

In Pascal gibt es zwei Standarddateien `input` und `output`, die beide vom Standarddatentyp `text` sind[28].

9.1.1 Textdateien und die Standarddateien `input` und `output`

Der vordefinierte Typbezeichner `text` stellt einen speziellen Dateityp dar, der dazu dient, um eine Folge von Zeichen aufzunehmen, die in verschieden lange **Zeilen** unterteilt sein kann. Eine Datei vom Typ text ist also in null oder mehr Zeilen strukturiert. Eine Zeile enthält null oder mehr Zeichen (Werte vom Typ `char`), gefolgt von einem speziellen **Zeilenendezeichen**. Eine Variable vom Typ Text wird **Textdatei** genannt.

Es gibt verschiedene Standardprozeduren und -funktionen zur Bearbeitung von Textdateien. An dieser Stelle sind hiervon interessant::

```
read
readln
write
writeln
eoln
eof
```

Die Standarddateien `input` und `output`

Die Standarddateien `input` und `output` stellen normalerweise die Standard-Ein- und Ausgabemedien eines Computersystems (wie Tastatatur und Bildschirm) dar. Deshalb

[28] Die möglichen Datentypen von Dateien werden an dieser Stelle nicht behandelt

sind sie der Hauptkommunikationsweg zwischen dem Computersystem und dessen menschlichem Benutzer.

Da diese beiden Dateien sehr oft verwendet werden, werden sie in den Textdatei-Operationen als voreingestellte Werte angesehen, wenn die Textdatei `input` bzw. `output` nicht als Parameter erscheint. Das heißt

```
write (p1, p2... pn)     = write (output, p1, p2... pn)
writeln (p1, p2... pn)   = writeln (output, p1, p2... pn)
read (v1, v2... vn)      = read (input, v1, v2... vn)
readln (v1, v2... vn)    = readln (input, v1, v2... vn)
```

Hierbei können

`p1, p2... pn` **Ausdrücke** vom Typ `integer`, `real`, `char`, `boolean` oder eine Zeichenkette sein.[29]

`v1 v2... vn` Variablen vom Typ `integer, real, char` sein.[30]

Mit `write` bzw. `writeln` werden Werte des Programms ausgegeben. Werte können **beliebige Ausdrücke** sein.

Mit `read` bzw. `readln` werden Tastatureingaben in **Variablen** des Programms abgelegt.

Auch bei den Standardfunktionen `eof` und `eoln` ist die Angabe von `input` nicht erforderlich.

```
eof              = eof (input)
eoln             = eoln (input)
```

Die Dateien, mit denen ein Programm Daten austauscht, sind im Programmkopf aufzulisten.

Beispiel:

```
program eins (input, output);
```

Bei Turbo Pascal kann man - wie bereits erwähnt - auch schreiben:

```
program eins;
```

Zeilenstruktur von `input`

1. Zeile ◆ 2.Zeile ◆ ... ◆ letzte Zeile ◆

[29] in Turbo-Pascal auch vom Typ `string`

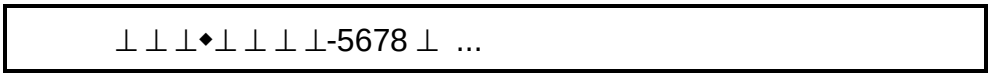
- Zeilenwechsel
  eof-File-Marke

Für die Funktion des Lesens und Schreibens muß man sich einen Dateizeiger vorstellen. Dieser wird in Abbildungen durch dargestellt. Der Dateizeiger zeigt auf die Stelle, an der gerade gelesen wird. Um die Wirkung beim Lesen besser zu zeigen, bedeuten vor und nach die Position des Dateizeigers vor und nach einem `read`.

9.1.2 Lesen von `input`

Mit `read` bzw. `readln` können `integer`-Zahlen, `real`-Zahlen und Zeichen (`char`) in Variable eingelesen werden.

Read einer integer-Zahl



vor nach

Das Zeichen ⊥ bedeutet Blank.

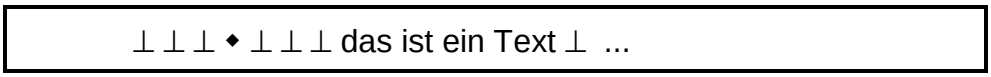
```
zahl: integer;

begin
   ...
   read (zahl);
   ...
```

Führende Zeichen Blank und Zeilenwechsel werden überlesen. Das erste gefundene Symbol wird als `integer`-Zahl interpretiert, konvertiert und der Variablen `zahl` zugewiesen. Steht an dieser Stelle keine `integer`-Zahl, so kommt es zu einer Fehlermeldung.

Read einer `real`-Zahl

Funktioniert wie das Lesen einer `integer`-Zahl.

Read eines Zeichens (`char`)



vor
nach

[30] in Turbo-Pascal auch vom Typ `string`

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
zeichen: char;

begin
   ...
   read (zeichen);
   ...
```

Hier wird das nächste Zeichen gelesen, unabhängig, was es bedeutet. **Zeilenwechsel** (unter MS-DOS zwei Zeichen: **Wagenrücklauf** (**Carriage-Return**, **CR**, ASCII dezimal 13) und **Neue Zeile** (**Line Feed**, **LF**, ASCII dezimal 10), Blanks etc. werden alle zeichenweise gelesen.

Das nächste `read` in eine Charakter-Variable bedient sich des nächsten Zeichens, etc.

Readln einer `integer`-Zahl, einer `real`-Zahl, eines Zeichens

Nach dem Einlesen in die entsprechende Variable wird der Rest der Zeile überlesen und der Dateizeiger auf den Beginn der nächsten Zeile positioniert.

Soll in mehrere Variablen von der Tastatur eingelesen werden, so kann als Argument der `read`-Funktion eine Liste von Variablen, die durch Kommas getrennt werden, angegeben werden. Stattdessen kann auch für jede einzelne Variable die `read`-Funktion aufgerufen werden.

```
read (v1, v2... vn)   =
read (v1); read(v2); ... ;read (vn)

readln (v1, v2... vn) =
read (v1); read (v2); ...;readln (vn)
```

9.1.3 Standardfunktionen `eoln` und `eof`

Im folgenden wird die Bedeutung der Standardfunktionen `eoln` und `eof` erläutert.

`eoln = true`,	wenn Dateizeiger auf Zeilenwechsel steht
`eoln = false`,	wenn Dateizeiger nicht auf Zeilenwechsel steht
`eof = true`,	wenn Dateizeiger das Dateiende erreicht hat
`eof = false`,	wenn Dateizeiger das Dateiende nicht erreicht hat

9.1.4 Schreiben nach `output`

Mit `write` bzw. `writeln` können **Ausdrücke** vom Typ `integer`, `real`, `char`, `boolean` oder Zeichenketten weggeschrieben werden.

Write einer `integer`-Zahl

Beispiel : p := -5678

```
⊥⊥⊥⊥⊥⊥⊥-5678
```

vor nach

Das Zeichen ⊥ bedeutet Blank.

Anmerkung: entsprechend der Zahl der Ausgabestellen des Standardformats für das Schreiben von `integer`-Zahlen werden beim Schreiben einer `integer`-Zahl führende Blanks erzeugt.

```
p: integer;

begin
   ...
   write (p);
   ...
```

Hat `p` einen Wert vom Typ `integer`, so wird der Wert konvertiert und in einer Standardform mit einer bestimmten Anzahl von Stellen weggeschrieben.

Write einer `real`-Zahl

Funktioniert wie das Schreiben von `integer`-Zahlen. Die Konvertierung erfolgt in eine Standardform für `real`-Zahlen, z.B. mit einer n-stelligen Mantisse und einem m-stelligen Exponenten.

Beispiel: p := -11.23

```
-1.1230000000E+01
```

vor nach

Write eines Zeichens (`char`)

Beispiel: p := 'g'

```
g
```

vor
nach

Write einer Zeichenkette

Beispiel: write ('FHTE')

FHTE

vor

nach

Writeln eines Ausdrucks vom Typ `integer`, `real`, `char` oder einer Zeichenkette

Nach dem Wegschreiben des entsprechenden Ausdrucks beziehungsweise der Zeichenkette, wird nach `output` ein Zeilenwechsel geschrieben.

Die Anweisung `writeln` ohne Parameterliste erzeugt einen Zeilenwechsel.

Schreibweisen:

```
write (p1, p2... pn)   = write (p1); write (p2); ...;write (pn)
writeln (p1, p2... pn) = write (p1); write (p2); ...;writeln (pn)
```

Formatierung

Für das Schreiben mit `write` bzw. `writeln` sind Formatangaben möglich. Ist der Ausdruck pi vom Typ `integer`, `char`, `boolean` oder eine Zeichenkette, so kann mit Hilfe des Ausdrucks d vom Typ `integer` durch

```
write (pi:d)
```

die Breite der Ausgabe festgelegt werden. Hierbei bedeutet der Wert von d die Anzahl der auszugebenden Zeichen. Beachten Sie hierbei, dass wenn die Zahl 3 vier Zeichen breit ausgegeben wird, der Dreier rechtsbündig steht und davor drei Leerzeichen sind.

Ist pi vom Typ `real`, so kann man mit Hilfe der Ausdrücke d und s jeweils vom Typ `integer` durch

```
write (pi:d:s)
```

die gesamte Zahl der Ausgabestellen durch d und die Zahl der Stellen hinter dem Dezimalpunkt durch s festlegen.

9.1.5 Beispiele für die Ein- und Ausgabe

Beispiel 1:

```
program char_int_io (input, output);
```

```
var  a:  integer;
     b:  char;

begin
   writeln;
   write ('Bitte eine Zahl (0-9) eingeben:');
   readln (a);
   write ('Bitte einen Buchstaben (A-Z) eingeben');
   read (b);
   write ('Sie haben gerade eine ', a:6, ' und ein ',b:6);
   writeln (' eingegeben');
end.
```

Beispiel 2:

Das folgende Programm hat einen Fehler. Trotz Eingabe von `N` bleibt das Programm stets in der `repeat`-Schleife. Was ist der Grund?

```
program  N_Fakultaet;

var    i  : integer;
       n  : integer;
       p  : real;
       ch : char;

begin
  repeat
       write ('Eingabe: N =');
       read(n);
       i:=0;
       p:=1;
       while i< n do
         begin
            i:=i+1;
            p:=p*i;
         end;
       writeln ('n-Fakultät = ',p);
       writeln;
       writeln ('Noch einmal? [J/N]:  ');
       readln(ch);
       writeln;
  until (ch='N')
end.
```

10 Lektion 10 - Unterprogramme

In den Übungen zum letzten Kapitel wurden die Programme schon größer und unübersichtlicher. Der Sinn einer guten Programmiersprache ist aber, dass man die Aufgaben innerhalb eines Programms verteilt und dass die einzelnen Funktionsblöcke übersichtlich bleiben. Dazu verwendet man **Unterprogramme**. Während es beispielsweise in der Programmiersprache C als Unterprogramme nur Funktionen gibt, hat Pascal **Prozeduren** und **Funktionen**.[31]

Auf die Möglichkeit Unterprogramme in Unterprogrammen zu schreiben, wird in diesem Grundkurs nicht eingegangen.[32]

Unterprogramme können innerhalb eines Programmes öfter aufgerufen werden. Diese **Wiederverwendbarkeit** ist ein zentrales Merkmal des Software-Engineerings

Durch die Verwendung von Unterprogrammen wird der Quellcode kleiner, als wenn die im Unterprogramm verwendete Funktionalität an mehrere Stellen hinkopiert wird. Das Programm wird dadurch **übersichtlicher.**

Dadurch, dass diesselbe Funktionalität nur einmal da steht, sind Änderungen leichter durchzuführen. Müssten Änderungen an vielen Stellen durchgeführt werden, so würde leicht eine Stelle übersehen werden und sich dadurch ein Fehler einschleichen. Das Programm wird **weniger fehleranfällig**.

Unterprogramme dienen also dazu, mehrere Anweisungen zusammenzufassen, die dann unter dem Namen des Unterprogramms aufgerufen werden können. Bis jetzt wurden "unbewusst" schon die Ein-/Ausgabe**prozeduren** `write`, `writeln`, `read` und `readln` verwendet. Bei den Zeichen wurden schon die Umwandlungs**funktionen** `ord` und `chr` verwendet. Diese Prozeduren und Funktionen sind in der Sprache schon enthalten und sind **Standardprozeduren** bzw. **Standardfunktionen**.

Zur Verdeutlichung von Unterprogrammen ein kurzes Programm, das einen Buchstaben einliest und den entsprechenden ASCII-Wert angibt.

```
program ascii1

var buchstabe : char;
var zahl : integer;

begin
  write('Bitte einen Buchstaben eingeben ');       {Prozedur}
  readln(buchstabe);                               {Prozedur}
  zahl := ord (buchstabe);                         {Funktion}
  writeln( buchstabe , 'entspricht dem Wert : ' ,
           zahl);                                  {Prozedur}
end.
```

[31] Hier darf man sich nicht verwirren lassen, denn der Umfang der Funktionen in C deckt den Umfang von Funktionen + Prozeduren in Pascal ab.

[32] Die Programmiersprache C bietet diese Möglichkeit nicht.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Eine ausführliche Programmbeschreibung könnte so aussehen:

- Das Programm ascii1 gibt zuerst in der **Prozedur** `write()` eine Zeichenkette auf den Bildschirm aus.
- Dann liest die **Prozedur** `readln()` ein Zeichen in die Variable `buchstabe` ein.
- Als nächstes wandelt die **Funktion** `ord()` den Inhalt der Variablen `buchstaben` in eine Zahl um und weist diese der Variablen `zahl` zu.
- Zuletzt gibt die **Prozedur** `writeln()` das Ergebnis auf dem Bildschirm aus.

Worum es jetzt geht, ist, **eigene** Prozeduren und Funktionen zu schreiben.

Wenn man eine Prozedur oder Funktion aufrufen will, muss diese erst vereinbart sein. Werden Prozeduren oder Funktionen aufgerufen, die dem Compiler nicht bekannt sind, so gibt es einen Kompilierfehler, denn etwas Unbekanntes kann man natürlich nicht verwenden.

Der Vereinbarungsteil eines Programms endet mit der Vereinbarung der Funktionen und Prozeduren. Eine Funktion oder Prozedur darf im Anweisungsteil erst benutzt werden, wenn sie im Vereinbarungsteil deklariert wurde.

10.1 Aufruf von Funktionen

Funktionen in Programmiersprachen funktionieren so, wie man es von der Mathematik her kennt.

Wenn man in der Mathematik schreibt

y = sin x

so kann man das in der folgenden Weise interpretieren:

Übergebe der Funktion sin den Wert von x. Die Funktion sin x hat einen Wert und dieser Wert wird y zugewiesen.

Genauso geht es auch beim Programmieren. Nur ist die Schreibweise etwas anders

```
y := sin (x)
```

Der Wert, der an die Funktion übergeben wird, wir innerhalb einer runden Klammer übergeben. Ferner wird in Pascal der Zuweisungsoperator durch `:=` ausgedrückt.

x heisst aktueller Parameter der Funktion sin(x). Ein aktueller Parameter wird beim Aufruf übergeben.

Funktionen können in Ausdrücken verwendet werden wie in folgendem Beispiel:

y:= 3 * sin (x)
Ausdruck

Wenn der Name einer Funktion in einer Anweisung auftaucht, dann unterbricht der Rechner die Bearbeitung des aufrufenden Programms und bearbeitet zunächst die Funktion. Ist der Wert der Funktion ermittelt, wird er an die Stelle des Aufrufs in den entsprechenden Ausdruck eingesetzt. Die Bearbeitung des aufrufenden Programms geht dann an eben dieser Stelle weiter, wo es ja auch vorher unterbrochen wurde.

10.2 Vereinbarung von Funktionen

Wir wollen jetzt eine Funktion zum Quadrieren schreiben. Der Aufruf der Funktion soll erfolgen durch

```
quadrat (x)
```

x soll dabei ein `integer`-Wert sein.

Die Vereinbarung der Funktion hat im Vereinbarungsteil vor dem Hauptprogramm zu erfolgen. Die Vereinbarung sieht folgendermaßen aus:

```
function quadrat (a : integer) : integer {Funktionskopf}

begin
    quadrat := a * a;
end
```

Funktionsrumpf

Der **Funktionskopf** hat 3 Aufgaben:

1. Er muss sagen, dass es sich um eine Funktion handelt. Dies erfolgt durch das Schlüsselwort `function`
2. Der Name der Funktion muss festgelegt werden. Dies erfolgt durch den Bezeichner nach dem Schlüsselwort `function`
3. Die Schnittstellen der Funktion zum Aufrufer hin zu beschreiben

In diesem Beispiel gibt es als Schnittstellen:

- den Übergabeparameter
- den Rückgabewert

Der Typ des Rückgabewerts ist `integer`. Dies wird durch

```
: integer
```

im Funktionskopf zum Ausdruck gebracht.

Wenn beim Aufruf

```
quadrat (x)
```

der `integer`-Wert `x`

übergeben werden soll, so muss für die Aufnahme dieses Wertes eine lokale Variable in der Funktion zur Verfügung stehen, die den Wert von `x` übernimmt.

Diese Variable in der Funktion ist der **formale Übergabeparameter** `a.` Der formale Übergabeparameter ist eine Variable, die nur innerhalb der Funktion `quadrat` sichtbar ist und nur innerhalb der Funktion benutzt werden kann.

Beim Aufruf der Funktion findet vollautomatisch ein Kopieren des Wertes des aktuellen Parameters `x` in die Variable a statt:

```
a := x
```

Zu jedem **aktuellen Parameter** des Aufrufs einer Funktion muss es einen **formalen Parameter** geben. Der Typ des formalen Parameters, der tatsächlich eine lokale Variable der Funktion ist, muss mit dem Typ des aktuellen Parameters übereinstimmen. Die einzige Ausnahme ist: der aktuelle Parameter ist vom Typ `integer`, dann kann der formale Parameter auch vom Typ `real` sein.

Der formale Parameter **muss** eine Variable sein. Der aktuelle Parameter **muss** ein Ausdruck sein. Eine Variable stellt auch einen Ausdruck dar, d.h. der aktuelle Parameter **kann** eine Variable sein.

Funktionsrumpf

Innerhalb des Funktionsrumpfes kann man dann mit der Variablen a, die den Wert von `x` enthält, weiterrechnen.

Was wird hier berechnet ?

Berechnet wird der Wert von

```
x*x
```

Dieser Wert wird dann beim Aufruf der Funktion durch

```
y := quadrat (x)
```

der Variablen y, die natürlich vom Typ `integer` sein muss, zugewiesen wird. Der Wert der zugewiesen wird, heisst **Rückgabewert**.

Eine Funktion hat einen Wert, der ihr Ergebnis darstellt. Das Ergebnis der Funktion wird als ihr **Rückgabewert** bezeichnet.

Wie der Rückgabewert zur Laufzeit übergeben wird, hängt vom Compiler ab. Am besten stellt man sich das folgendermaßen vor:

Zunächst wird in eine temporäre Variable und für den Programmierer unsichtbare Variable kopiert. Dies erfolgt durch die Zuweisung

```
quadrat := x*x
```

Man kann sich also vorstellen, dass diese temporäre und für den Programmierer unsichtbare Variable sich hinter dem Funktionsnamen verbirgt. Der zweite Kopiervorgang ist dann

```
y:= quadrat
```

Eine Funktion kann **mehrere Übergabeparameter** haben. Mit jedem Übergabeparameter wird ein Wert an die Funktion übergeben.

Eine Funktion kann nur **einen einzigen** Wert zurückgeben. Dies ist ihr **Rückgabewert**.

Möchte man mehrere Werte zurückgeben, so braucht man dazu Prozeduren.

10.3 Aufruf von Prozeduren

Eine Prozedur hat keinen Rückgabewert wie eine Funktion. Da sie nichts zurückgibt, kann sie also nicht in einer Zuweisung verwendet werden. Eine Prozedur wird durch

```
proc (a, b)
```

aufgerufen. Dann ist eine Zuweisung

```
y:= proc (a,b)
```

total unsinnig, da auf der rechten Seite der Zuweisung ja ein Ausdruck vom Typ der Variablen stehen müsste. Der Aufruf einer Prozedur stellt also stets eine eigene Anweisung dar.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
....;
proc (a, b);
....;
```

Prozeduren werden durch eine eigene Anweisung, die **Prozeduranweisung**, aufgerufen. Funktionen können **innerhalb von Ausdrücken** aufgerufen werden.

Das Besondere einer Prozedur ist außer dem fehlenden Rückgabewert, dass Prozeduren direkt auf Variablen, die ihnen übergeben wurden, arbeiten und diese Variablen verändern können.

Natürlich muss man diese Art von Übergabeparamatern besonders kennzeichnen. Die Kennzeichnung erfolgt im Kopf der Prozedur, indem dem entsprechenden Übergabeparameter das Schlüsselwort `var` vorangestellt wird.

Im folgenden wollen wir eine Prozedur `quadrat` verwenden:

Hier das vollständige Programm:

```
programm quadratprozedur;

var zuquadrierendeZahl : integer;
    quadratzahl        : integer;

procedure quadrat (var ergebnis: integer; zahl: integer)

begin
    ergebnis:= zahl * zahl;
end

begin                          {dies ist das Hauptprogramm}
     writeln ('Quadrat von welcher Zahl ?');
     readln (zuquadrierendeZahl);
     procedure (quadratzahl, zuquadrierendeZahl),
     writeln (quadratzahl, 'ist das Quadrat von ',
              zuquadrierendeZahl);
end.
```

Eine **Prozedur** kann wie eine Funktion mehrere Eingangsgrößen haben. Eine Prozedur kann auch mehrere Ausgangsgrößen haben. Eingangs- und Ausgangsgrößen werden bei der Prozedur durch die Parameterliste, die Eingangs- und Ausgangsgrößen enthalten kann, bereit gestellt.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Hat eine Prozedur oder eine Funktion keine Parameter, so fehlt die Parameterliste ganz, d.h. dass auch die runden Klammern fehlen.

Im Falle einer **Funktion** können die Eingangsgrößen ebenfalls über die Parameterliste übergeben werden. Es gibt bei der Funktion jedoch nur eine einzige Ausgangsgröße. Diese wird bei der Funktion nicht über die Parameterliste übergeben, sondern durch die Zuweisung des Funktionswertes zu einer Variablen

Werteparameter und Referenzparameter

Bei formalen Parametern ohne das Schlüsselwort `var` wird beim Aufruf der Wert des aktuellen Parameters in den formalen Parameter kopiert. Aktueller und formaler Parameter sind ansonsten vollkommen unabhängig voneinander. Änderungen am formalen Parameter beeinflussen den aktuellen Parameter nicht. Ein solcher formaler Parameter heisst **Werteparameter**.

Wird das Schlüsselwort `var` verwendet, so sind formaler und aktueller Parameter nicht entkoppelt. Eine jede Änderung des formalen Parameters erfolgt tatsächlich auf dem aktuellen Parameter[33]. Ein solcher formaler Parameter heisst **Referenzparameter**.

Eine jede Operation auf dem Referenzparameter erfolgt tatsächlich auf der referenzierten Variablen, dem aktuellen Parameter

Mit Referenzparametern hat man deshalb nicht nur die Möglichkeit, von der aufrufenden Routine der aufgerufenen Prozedur einen Wert zu übergeben, sondern von der aufgerufenen Prozedur auch einen Wert zurückzuerhalten.

Beispiel für Referenz- und Werteparameter:

```
procedure demo (var i: integer; c: char;
                var p, q, w : real);
```

Diese Prozedur hat die Referenzparameter: `i,p,q,w`
und den Werteparameter: c

[33] dieser muss auch zwingend eine Variable sein und kein allgemeiner Ausdruck wie z.B. 3*a + b

Verwendung von Referenz- und Werteparametern:

- Ausgangsgrößen müssen bei Prozeduren immer Referenzparameter sein, weil man anders keine Werte aus einer Prozedur hinausbringt.
- Bei Eingangsgrößen hat man die Wahl zwischen Referenz- und Werteparametern.
- Bei Werteparametern kann man Ausdrücke einsetzen, bei Referenzparametern nur Namen von Variablen.
- Bei Werteparametern bearbeitet die Prozedur Kopien der aktuellen Werte, bei Referenzparametern das Original.
- In der Praxis werden Eingangsparameter meist Werteparameter sein.

Formale Parameter dienen zur Beschreibung einer Prozedur oder Funktion. Beim Aufruf wird der formale Parameter durch den aktuellen Parameter ersetzt. Ein formaler Parameter ist ein Platzhalter.

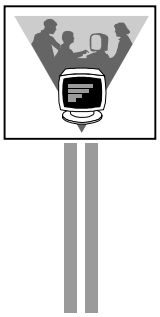
Die aktuellen Parameter von Prozeduren oder Funktionen müssen in Anzahl, Reihenfolge und Typ mit den formalen Parametern übereinstimmen.

Die Namen der formalen Parameter können völlig frei vereinbart werden. Sie sind nur lokal in der jeweiligen Prozedur sichtbar. Der formale Parameter kann denselben Namen wie der aufrufende Parameter haben, muß es aber nicht. Es kann übersichtlicher sein, wenn die Namen verschieden sind.

Beispiel:

```
program Parameter_Mechanismus (input, output);

var    i    :    integer;
       k    :    integer;
       x    :    integer;
       y    :    integer;


procedure zaehl_per_wert_aufruf (a: integer);

begin
   a :=  a + 1;
end;

procedure zaehl_per_reference_aufruf (var a: integer);

begin
   a :=  a + 1;
end;

procedure zaehl (a: integer; var b: integer);
```

```
begin
  b:= a + 1;
end;


begin (* Anweisungsteil *)
      writeln ('Gib drei Integer-Zahlen ein: ');
      readln (i, k, x);

      writeln ('i vor zaehl_per_wert_aufruf:', i:5);
      zaehl_per_wert_aufruf (i);
      writeln ('i nach zaehl_per_wert_aufruf:', i:5);

      writeln ('k vor zaehl_per_reference_aufruf:', k:5);
      zaehl_per_reference_aufruf (k);
      writeln ('k nach zaehl_per_reference_aufruf:',k:5);

      writeln ('x vor zaehl:', x:5);
      zaehl (x,y);
      writeln ('y nach zaehl:', y:5);
end.
```



Wenn man das Programm startet und die Eingabe z.B. mit den Zahlen 8 8 8 bedient, ergibt sich die Ausgabe:

```
i vor zaehl_per_wert_aufruf:    8
i nach zaehl_per_wert_aufruf:    8
k vor zaehl_per_refence_aufruf:    8
k nach zaehl_per_refence_aufruf:    9
x vor zaehl:    8
y nach zaehl:    9
```

Beispiel:

```
Program Durchschnitts_Berechnung (input, output);

var Mittelwert   : real;
    Zahl_1       : real;
    Zahl_2       : real;

function berechne_Mittel (a, b : real): real;
(* beachte Komma in formaler Parameterliste *)
(* warum Komma ?                           *)
begin
   berechne_Mittel:= (a + b)/2.0;
end;

procedure eingabe (var x: real; var y: real);
(* beachte Strichpunkt in formaler Parameterliste *)
(* warum Strichpunkt ?                             *)
begin
   writeln;
   write ('Gib die erste Zahl ein [real]:      ');
```

```
   read (x);
   write ('Gib nun die zweite Zahl ein [real]: ');
   read (y);
end;

procedure ausgabe (x,y,z: real);
(* beachte Kommata in formaler Parameterliste *)
(* warum Kommata ?                            *)

begin
   writeln;
   writeln ('Der Mittelwert von ',x:10:2);
   writeln ('und                ',y:10:2);
   writeln ('ist:               ',z:10:2);
end;

begin
  eingabe (zahl_1, zahl_2);
  Mittelwert:= berechne_mittel (zahl_1, zahl_2);
  ausgabe (zahl_1, zahl_2, Mittelwert);
end.
```

Weitere Beispiele:

In Pascal gibt es keine Standardfunktion für das Potenzieren "x hoch n". Diese Berechnung kann mit einem Unterprogramm realisiert werden, das für ein ganzzahliges positives n die n-te Potenz von x berechnet. Als Anwendung soll $a^b + c^d$ berechnet werden.

Zuerst eine Lösung mit einer Prozedur:

```
procedure hoch(x : integer; n : integer; var potenz : integer);
var i: integer;
begin
  if 0 = n then
    potenz := 1
  else
    begin
      potenz := 1;
      for i := 1 to n do
        potenz := potenz * x;
    end;
end;
```

Für die Berechnung braucht man hier drei Anweisungen und zwei zusätzliche Hilfsvariablen a_hoch_b und c_hoch_d.

```
hoch ( a, b, a_hoch_b);
hoch ( c, d, c_hoch_d);
ergebnis := a_hoch_b + c_hoch_d;
```

... und im Vergleich mit einer Funktion:

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
function hoch(x : integer; n : integer) : integer;
var i: integer, potenz;
begin
  if 0 = n then
    potenz := 1
  else
    begin
      potenz := 1;
      for i := 1 to n do
        potenz := potenz* x;
    end;
  hoch := potenz;
end;
```

Die Berechnung erfolgt hier in einer einzigen Anweisung:

```
ergebnis := hoch (a, b) + hoch (c, d);
```

10.4 Die Direktive forward

Generell kann in Pascal ein Name erst dann verwendet werden, nachdem er definiert bzw. vereinbart ist. Das bedeutet insbesondere, daß wenn eine Funktion oder Prozedur eine Funktion oder Prozedur aufruft, daß diese bereits vereinbart sein muß.

Beispiel:

```
procedure alt (z: real);
begin
  ...
end;

procedure neu (x:real);
begin
   ...
   alt(x);
   ...
end;
```

Nicht zugelassen ist:

```
procedure neu (x:real);
begin
   ...
   alt(x);
   ...
end;

procedure alt (z: real);
```

da die Prozedur `alt` noch nicht vereinbart ist, wenn sie in der Prozedur `neu` aufgerufen wird.

Mit Hilfe der Direktive **forward** kann man jedoch den Prozedurkopf und damit die Schnittstelle zur Prozedur, d.h. den Aufruf der Prozedur, dem Compiler bekannt machen. Die Implementation des Blocks der Prozedur (des Rumpfes der Prozedur) kann dann auch später noch erfolgen. Um in obigem Beispiel also die Prozedur alt benutzen zu können, bevor sie definiert wird, kann man durch die Direktive forward, einen sog. **Vorwärtsbezug**, den Prozedurkopf `alt` anführen und damit angeben, daß der zugehörige Block später im Programm folgt.

Beispiel:

```
procedure alt (z: real); forward;

procedure neu (x:real);
begin
   ...
   alt(x);
   ...
end;
```

Bevor dann der Block der mit `forward` aufgeführten Prozedur folgt, ist ihm die Angabe

```
procedure Prozedurname;
```

voranzustellen. Die Parameterliste (und bei Funktionen die Angabe über den Typ des Funktionswertes) stehen beim Vorwärtsbezug.

```
procedure alt; (* hier entfällt jetzt die Parameterangabe *)
begin
  ...
end;
```

Wenn `forward` verwendet wird, steht die Parameterliste und bei Funktionen die Angabe über den Typ des Funktionswertes beim Prozedurkopf vor der Direktive `forward`.
Dem Block ist nur noch `procedure` Prozedurname bzw. `function` Funktionsname voranzustellen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

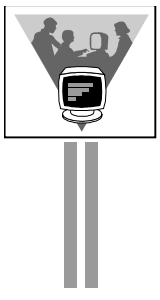
Ruft eine Prozedur oder eine Funktion sich selbst auf, spricht man von einer **direkten Rekursion**. Rufen sich Prozeduren gegenseitig auf, so spricht man von einer **indirekten Rekursion**. In diesem Kurs wird die Rekursion nicht behandelt.

10.5 Lokale und globale Variablen

In Kapitel 3.3 wurde bei den Variablen die Deklaration am Programmanfang besprochen. Diese Variablen sind im gesamten Programm gültig. Zusätzlich gibt es noch lokale Variablen. Sie sind nur in dem Bereich gültig, in dem sie deklariert sind. Dieser Bereich kann eine Prozedur oder eine Funktion sein.

Das folgende Beispiel verdeutlicht dies:

```
program lokalglobal;

var globales_x : integer;

procedure hallo[34];
  var lokales_x : integer;

  begin
    lokales_x := 20;
    writeln('  In der Funktion hallo');
    writeln('  Das globale x ist: ', globales_x);
    writeln('  Das lokale x ist : ', lokales_x);
  end;

begin
  globales_x := 10;
  writeln('Im Hauptprogramm');
  writeln('Das globale x ist: ', globales_x);
  hallo;
end.
```



Die zugehörige Ausgabe ist:

```
im Hauptprogramm
Das globale x ist: 10
  In der Funktion hallo()
  Das globale x ist: 10
  Das lokale x ist : 20
```

Die globale Variable `globales_x` ist überall gültig. Die lokale Variable `lokales_x` ist nur in der Prozedur `hallo` gültig.

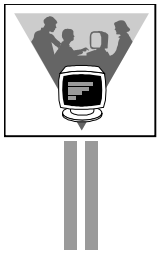
[34] Da die Prozedur `hallo` keine Parameter hat, entfallen die runden Klammern. Diese Ausnahme in Pascal verwirrt, da dadurch Prozeduren nicht mehr von Variablen unterschieden werden können!

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Haben eine globale und eine lokale Variable den gleichen Namen, wird die globale Variable durch die lokale verdeckt.

Das folgende Programm zeigt diesen Zusammenhang auf:

```
program lokal2;

var x : real;  {globales x}

procedure hallo[35];
  var x : integer;  {lokales x}

  begin
    x := 20;
    writeln('  In der Funktion hallo');
    writeln('  Das lokale x ist : ', x);
    x := x + 2;
    writeln('  Das lokale x ist : ', x);
  end;

begin
  x := 10.5
  writeln('Im Hauptprogramm');
  writeln('Das globale x ist: ', x:3:1);
  hallo;      {Unterprogramm-Aufruf}
  writeln('Das globale x ist: ', x:3.1);
end.
```



Die Ausgabe ist hier:

```
Im Hauptprogramm
Das globale x ist: 10.5
  In der Funktion hallo
  Das lokale x ist : 20
  Das lokale x ist : 22
Das globale x ist: 10.5
```

Das globale x ist während der Prozedur `hallo` unsichtbar. Es spielt hier keine Rolle, dass die Variablen x unterschiedliche Typen haben.

10.6 Aufrufhierarchie und Unterprogrammreihenfolge

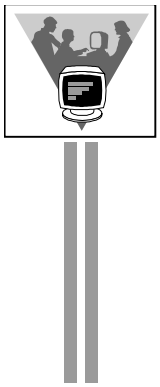
Unterprogramme können ihrerseits wieder Unterprogramme aufrufen. Dadurch entstehen Aufrufhierarchien.

[35] Da die Prozedur `hallo` keine Parameter hat, entfallen die runden Klammern. Diese Ausnahme in Pascal verwirrt, da dadurch Prozeduren nicht mehr von Variablen unterschieden werden können!

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Das aufgerufene Unterprogramm muss vor dem aufrufenden Programmteil (Hauptprogramm, Unterprogramm) deklariert sein.

```
program aufruf1;

procedure up3;
begin
  writeln('      procedure 3 wird gestartet');
  writeln('      procedure 3 wird beendet');
end;

procedure up2;
begin
  writeln('    procedure 2 wird gestartet');
  up3;
  writeln('    procedure 2 wird beendet');
end;

procedure up1;
begin
  writeln('  procedure 1 wird gestartet');
  up2;
  writeln('  procedure 1 wird beendet');
end;

begin
  writeln('Hauptprogramm wird gestartet');
  up1;
  writeln('Hauptprogramm wird beendet');
end.
```



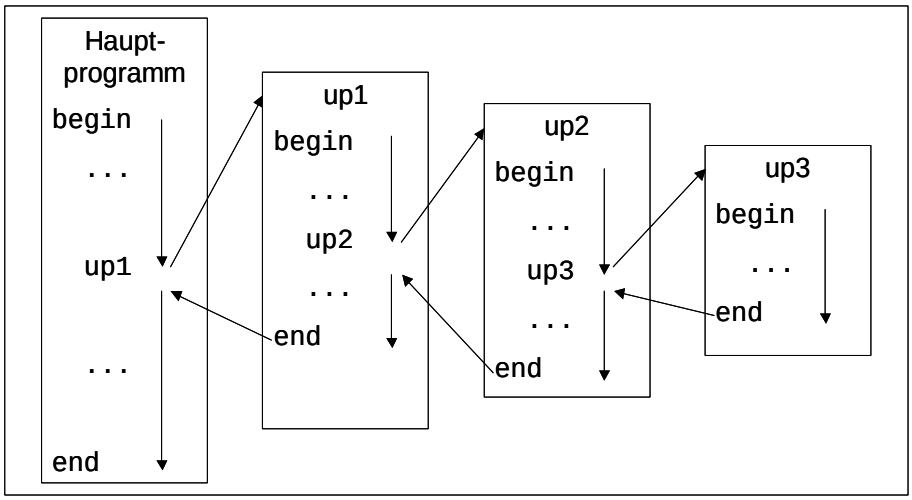
Die Ausgabe ist:

```
Hauptprogramm wird gestartet
  procedure 1 wird gestartet
    procedure 2 wird gestartet
      procedure 3 wird gestartet
      procedure 3 wird beendet
    procedure 2 wird beendet
  procedure 1 wird beendet
Hauptprogramm wird beendet
```

Man sieht schön, wie die einzelnen Prozeduren sich gegenseitig aufrufen. Den Programmfluss kann man sich so vorstellen:

Bild 10-1Programmfluss für `aufruf1`

Die Programmaufrufe kann man sich an einer Aufrufhierarchie graphisch



veranschaulichen:

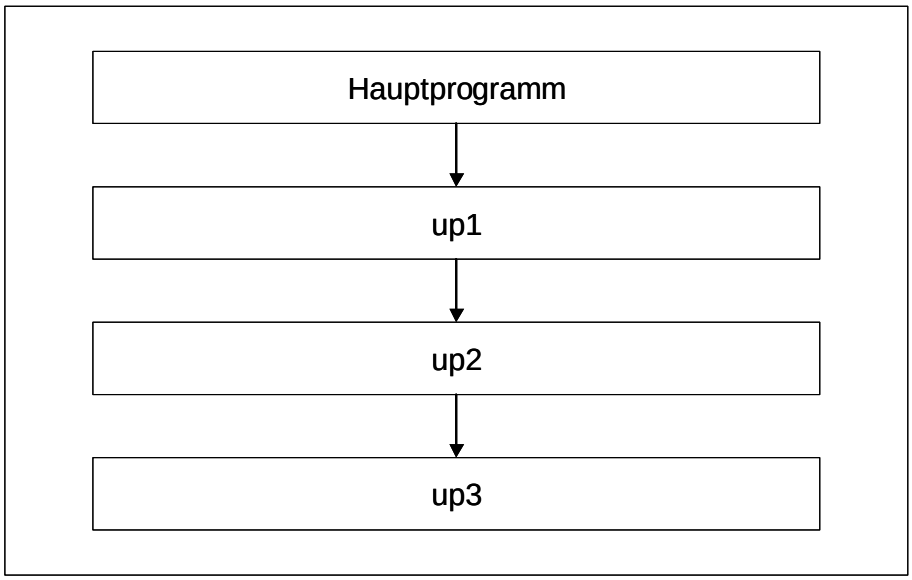


Bild 10-2 Aufrufhierarchie von aufruf1

10.7 Aufgaben

Aufgabe1: Die Potenz x^n

Ergänzen Sie die fehlenden Teile der folgenden Funktion. Fehlende Teile sind gekennzeichnet durch
`. . . . . .`

```
function hoch(x : integer; n : integer) : integer;
```

```
var i: integer, potenz;
begin
  if 0 = n then
    potenz := 1
  else
    . . . . .
end;
```

Aufgabe 2: Globale und lokale Variable

Welche Zahlenwerte werden im folgenden Programm ausgegeben ?

```
Die globalen Variablen haben den Wert x:    ?    y:    ?    z:    ?
innerhalb alpha                        x:    ?    y:    ?    z:    ?
```

Setzen Sie von Hand die Zahlen ein. Implementieren Sie anschließend das Programm!

```
program globale_und_lokale_variable (input,output);
var x, y, z: integer;

(* Beginn alpha *********************************)
procedure alpha (y: integer);

var x: integer;

begin
   x:= y;
   y:= y * y;
   writeln ('innerhalb alpha                    ','    x: ',
             x: 4, '    y: ', y: 4,'    z: ', z: 4);
end;
(* Ende alpha ***********************************)


begin
   x:= 2;
   y:= 4;
   z:=6;
   writeln;
   writeln ('Die globalen Variablen haben den Wert x: ',
             x:4, '    y: ', y:4, '    z: ', z:4);
   alpha (y);
end.
```

Programm 10-1 globale und lokale Variablen

Aufgabe 3: Funktionen

Schreiben Sie ein Programm zur Berechnung des Mittelwerts von 2 Zahlen unter Verwendung von 2 Prozeduren und einer Funktion:

- der Prozedur **eingabe**
- der Funktion **berechne_mittel**
- und der Prozedur **ausgabe**

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Die Prozedur **eingabe** liest die beiden Zahlen vom Terminal ein, die Funktion **berechne_mittel** rechnet den Mittelwert aus und die Prozedur **ausgabe** gibt das Ergebnis am Terminal aus.

Der Anweisungsteil des Hauptprogramms sehe folgendermaßen aus:

```
begin
  eingabe (zahl_1, zahl_2);
  Mittelwert:= berechne_mittel (zahl_1, zahl_2);
  ausgabe (zahl_1, zahl_2, Mittelwert);
end.
```

Schreiben Sie das komplette Programm!

Aufgabe 4: Funktionen und Prozeduren

Entwickeln Sie ein Programm, bei dem solange ganze Zahlen eingeben werden, bis die Zahl 0 ist. Geben Sie abschliesend die größte und die kleinste Zahl, den Mittelwert und die Summe aller Zahlen aus.

Hinweis: Verwenden Sie zwei Variablen, die den größten und den kleinsten Wert beinhalten und vergleichen Sie in jeder Schleife die eingegebene Zahl mit diesen Variablen: Wenn die aktuelle Zahl größer als die bisher größte Zahl ist, dann ist die aktuelle Zahl die (neue) größte Zahl.

Welchen Wert müssen die Variablen für die größte/kleinste Zahl am Anfang haben?

Dieses Programm soll in mehreren Schritten mit Unterprogrammen geschrieben werden.
Benutzen Sie während der Programmerstellung `writeln()`-Anweisungen, um die Programmfunktionen zu testen. Diese Anweisungen werden im fertigen Programm wieder entfernt.

In diesem Programm wird die Bibliotheksfunktion clrscr (clear screen = Bildschirm löschen) aus der unit crt verwendet.

TIP: Dieses Programm ist umfangreicher. Zu zweit lässt es sich meistens schneller programmieren (einer denkt, der andere tippt ;-), nach jeder Teilaufgabe abwechseln ...

Der anfängliche Programmrumpf sieht so aus:

```
program zahlen;
uses crt; {Bibliothek}

{globale Variablen}
  var zaehler, summe : Integer;
  min, max, schnitt : Integer;
  aktuellerWert : Integer;

{Funktionen und Prozeduren}
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
{Hauptprogramm}
begin
  {Variablen initialisieren}
  zaehler := 0;
  summe := 0;
  min := 32000;
  max := -32000;

  {ab hier sollten hauptsächlich }
  {Unterprogramme aufgerufen werden.}
  begruessung
  ...
end.
```

a) Statischer Begrüßungstext als Prozedur ohne Parameter
Wie bei jedem professionellem Programm soll zuerst ein Begrüßungstext erscheinen, in dem der Programmname und die Programmfunktion vorgestellt wird. Der Benutzer soll durch ein Drücken der Eingabetaste (Entertaste) die Prozedur verlassen[36] Danach soll der Bildschirm wieder mit der (fertigen) Prozedur `clrscr` gelöscht werden. Entwerfen Sie die **Prozedur `begruessung`**, die vom Hauptprogramm aufgerufen wird.

b) Eingabe-Funktion für neuen Wert
Entwerfen Sie eine **Funktion `eingabe`**, in der nach Aufforderung neue Werte eingelesen werden. Der neue Wert wird als Rückgabewert der Funktion an das Hauptprogramm übergeben.

Der Eingabewert soll dabei in einer Schleife solange eingelesen werden, bis er 0 ist.

Programmcode für Hauptprogramm:

```
  ...
  aktuellerWert := eingabe;
  while (aktuellerWert <>0) do
  begin
    {Dieser Block wird in der nächsten Teilaufgabe ausgefüllt}

    aktuellerWert := eingabe; {naechste Eingabe}
  end;
  {wird demnaechst ersetzt...}
  writeln('Ausserhalb der Schleife');
  ...
```

Zusatz: Es sollen in der Funktion nur Werte kleiner als 1000 und größer als -1000 akzeptiert werden. Ansonsten kommt eine Fehlermeldung und die Eingabe wird wiederholt. (Tip: Repeat..Until-Schleife).

[36] Wird durch ein readln(chararkterVariable) realisiert.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

c) Prozedur für größten und kleinsten Wert
Als nächstes soll eine **Prozedur `minmax`** zur Berechnung des minimalen und maximalen Wertes geschrieben werden. Der Aufruf innerhalb der Schleife des Haupptrogramms soll so erfolgen:

```
minmax (aktuellerWert, min, max);
```

d) Funktion für Durchschnittsberechnung mit Wertparametern
Innerhalb der Schleife des Hauptprogramms soll bei jedem Durchlauf die Gesamtsumme der bisher eingegebenen Zahlen ermittelt werden. Außerdem soll ein Zähler ermitteln, wieviele Zahlen bisher eingegeben wurden.

```
summe := summe + aktuellerWert;
zaehler := zaehler + 1;
```

Nach dem Ende der Schleife soll die **Funktion `durchschnitt`** den Durchschnitt der bisher eingegebenen Zahlen ausgeben. Ist die Zählvariable 0, so soll 0 als Durchschnitt zurückgegeben werden.

Der Aufruf aus dem Hauptprogramm soll so erfolgen:

```
schnitt := durchschnitt (summe, zaehler);
```

e) Ausgabe des Ergebnisses als Prozedur mit Parameter
Schreiben Sie zum Schluss noch eine Ausgabe**prozedur**, mit der die untersuchten Werte

minimaler Wert,
maximaler Wert,
Anzahl der bearbeiteten Zahlen,
Summe aller Zahlen und
Durchschnitt aller Zahlen

ausgegeben werden. Die Werte sollen als Parameter vom Hauptprogramm übernommen werden. Löschen Sie vor der Ausgabe den Bildschirm mit der Prozedur `clrscr`.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

11 Lektion 11 - Eindimensionale Arrays

11.1 Einfache und zusammengesetzte Datentypen

Einfache Typen (Ordinaltypen und der `real`-Typ) sind unstrukturierte Typen. Die anderen Typen von Pascal sind **strukturierte Typen** und **Zeigertypen**.

Pascal kennt zwei Arten von **einfache**n **Datentypen**. Es sind die **Ordinaltypen und der Real-Typ**. Ordinaltypen sind Typen, bei denen die Elemente durchgezählt werden können und deren Anzahl endlich ist (siehe unten).

Die Ordinaltypen sind entweder vom Programmierer definiert und heißen dann Aufzählungs- oder Teilbereichstypen, oder es sind die durch einen der Bezeichner `boolean`, `integer` oder `char` repräsentierten ordinalen Standardtypen.

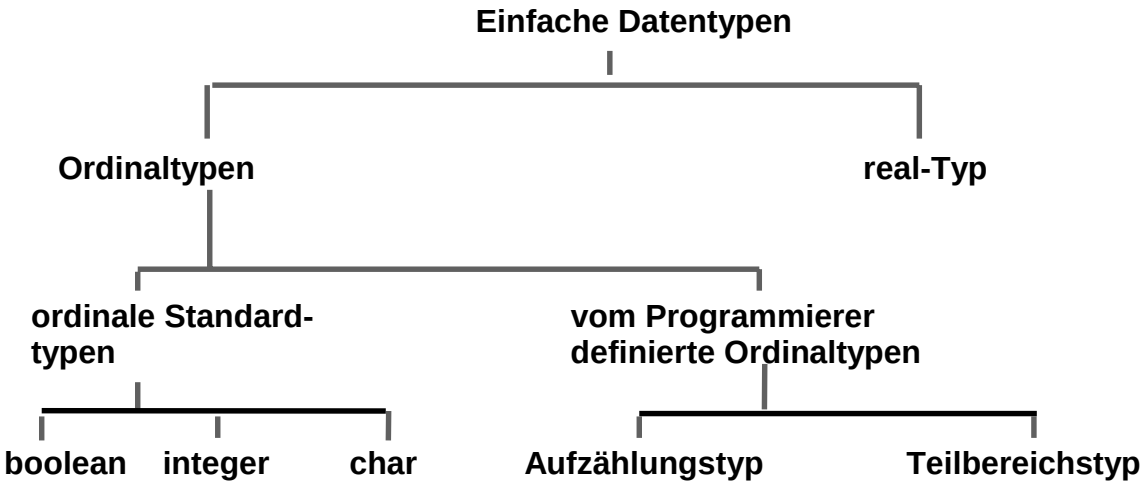


Bild 2.1 - 1 Übersicht über die einfachen Datentypen

In diesem Kurs werden vom Programmierer selbst definierte Ordinaltypen nicht behandelt, genau so wenig wie Zeigertypen.

Aus den einfachen Datentypen lassen sich komplizierter aufgebaute **strukturierte Datentypen** konstruieren. Die Typen der **Komponenten** und - das Wichtigste - **die Strukturierungsmethode** charakterisieren dabei den strukturierten Typ.

Die **strukturierten Datentypen** umfassen:

- Reihungs-Datentypen (Arrays)
- Verbund-Datentypen (Record)
- Mengen-Datentypen[37]
- Datei-Datentypen

[37] Der Mengentyp ist von untergeordneter Bedeutung und wird hier nicht behandelt.

Einfache Datentypen werden auch **elementare** Datentypen genannt. **Strukturierte** Datentypen werden auch als **zusammengesetzte** Datentypen bezeichnet.

Auf eine Variable eines strukturierten Typen kannn auf zweierlei Weise zugegriffen werden:

Zugriff über
- auf die Ganzvariable
- eine Komponentenvariable

Eine **Ganzvariable** ist eine Variable, die den **gesamten Speicherbereich** für einen einfachen oder strukturierten Typ oder einen Zeigertyp repräsentiert.

Auf die Komponente eines strukturierten Typs kann mit Hilfe von **Komponentenvariablen** zugegriffen werden, die den **Speicherbereich einer einzelnen Komponente** repräsentiert.

Für die Datenverarbeitung von besonderer Bedeutung sind die Typen Reihungstyp (Array-Typ), Verbundtyp und Dateityp.

Der **Reihungstyp** beschreibt Daten, die aus einer Anzahl gleicher Komponenten bestehen **(Array oder Reihung)**. In der Mathematik gehören hierzu Vektoren und Matrizen.

Der **Verbundtyp** beschreibt eine Datenstruktur **(Verbund oder Record)**, die aus einer gegebenen Anzahl nicht notwendig gleichartiger Komponenten besteht.

Was bei den zusammengesetzten Datentypen neu ist, dass man sie im Rahmen der Gegebenheiten beliebig zusammensetzen kann. Dies bedeutet, dass der Programmierer diese Datentypen im Gegensatz zu den einfachen Standarddatentypen wie `char`, integer, `real` oder `boolean` jetzt selbst definieren muss.

11.2 Der Arraytyp

Der **Arraytyp** besteht aus einer festen Anzahl von Komponenten (die bei der Definition des Reihungstyps festgelegt werden) vom selben Typ, dem **Komponententyp**.

Jede der Komponenten wird durch die **Array-Variable** mit anschließendem **Index in eckigen Klammern** repräsentiert. Über den Index kann direkt auf die Variable zugegriffen werden. Die Zeit, die benötigt wird, um auf eine Komponente zuzugreifen, hängt nicht vom Wert des Index ab. Deshalb wird ein Array auch als **Struktur mit wahlfreiem Zugriff (random access structure)** bezeichnet.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Indizes können Ausdrücke sein. Ihr Typ wird **Indextyp** genannt.

Arrays geben die Möglichkeit, mehrere Variablen mit gleichen Eigenschaften unter einem einzigen Namen zusammenzufassen. Durch die Vereinbarung einer Array-Variablen wird der ganzen Struktur ein Name gegeben.

Der Vorteil von Arrays gegenüber mehreren einfachen Variablen ist, dass Arrays sich leicht mit Schleifen bearbeiten lassen, da der Index einer Array-Komponente eine Variable sein kann und als Laufvariable in einer Schleife benutzt werden kann.

Mögliche Operationen bei Arrays

Operationen auf Ganzvariablen:
- **Auswahl (Selektion) von Komponenten (mit Operator [ ])**
- **Zuweisung**

Die Zuweisung und die Auswahl von Komponenten sind erlaubte **Operationen auf Array-Ganzvariablen.**

Die **Auswahl** einer Komponente erfolgt durch Angabe des Namens der Array-Variablen, gefolgt von dem Index in eckigen Klammern,wie zum Beispiel A[3]

Wenn A und B Array-Variablen vom selben Typ sind, dann ist die **Zuweisung**

```
A:= B
```

erlaubt, falls die Arrays komponentenweise zuweisbar sind:

```
A[i] := B[i]
```

(für alle `i`, deren Wert vom Indextyp ist). Diese Zuweisung ist eine Vereinfachung gegenüber der Zuweisung jeder der einzelnen sich entsprechenden Komponenten.

Operationen auf Komponentenvariablen:

- **alle Operationen, die für den Komponententyp des Reihungstyps gelten**

So müssen Arrays komponentenweise addiert werden, beispielsweise:

```
for LV := 1 to 6 do c[LV] := a[LV] + b [LV]
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

wobei `a, b` und c Arrays sind.

Ein Beispiel für eine Zuweisung ist:

```
Speicher [I + J] := X;
```

11.3 Vereinbarung von Arrayvariablen für eine eindimensionales `integer`-Array

Es soll eine Array-Variable `alpha` definiert werden, die 5 integer-Komponenten hat.

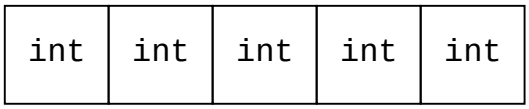


Bild 11-1 Ein Array aus 5 `integer`*-Elementen*

Hierfür gibt 2 Möglichkeiten:

Entweder:

```
var alpha = array [1..5] of integer;
```

Definition des
neuen Typs

oder

```
type intarray = array [1..5] of integer;
```

Typname

Typ

```
var alpha : intarray;
```

Allgmein ist der Array-Typ so definiert:

`array [``<von>..<bis>`**`] of`** `<Variablentyp>;`

Vor allem wenn man mehrere Variablen des geichen Arrays braucht, lohnt sich die Typdefinition, da dann immer nur der neue Typname anstelle der Typdefinition verwendet werden muss.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

11.4 Arrays als Parameter bei Prozeduren

Sind Arrays Parameter von Prozeduren oder Funktionen, so ist ihr Typname und nicht ihr Typ anzugeben, da die formale Parameterliste dies verlangt.

So muß es beispielsweise

```
type speicher = array [0..Max] of integer;
...
procedure proc (var x: speicher);
```

heißen und nicht

```
procedure proc (var x: array [0..Max] of integer);
```

11.5 Aufgaben

Aufgabe 1: Arrays

Weisen Sie einem Array aus 128 Zeichen Zeichen die Zeichen des ASCII-Zeichensatzes zu. Geben Sie die Zeichen mit der Ordinalzahl 48 bis 57 am Bildschirm aus.

Aufgabe 2: Arrays

Lesen Sie in ein Array aus 3 Integer-Komponenten von der Tastatur die Werte ein. Ermitteln Sie, welche Komponente den größten Wert hat und geben Sie die Nummer der Komponente und ihren Wert am Bildschirm aus.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

12 Lektion 12 - Records

Ein **Verbund** oder **Record** ist eine Struktur, die aus einer festen Anzahl von Komponenten, genannt **Felder** besteht. Im Unterschied zu einem Array können die Komponenten verschiedenartige Typen haben, können jedoch nicht durch einen Ausdruck indiziert werden. Die Typdefinition muß für jede Komponente deren Namen (**Feldbezeichner**) und Typ angeben.

In diesem Kurs werden nur Records mit festem, nicht mit variantem Anteil behandelt.

Die Bezeichnung Feld für eine Recordkomponente kommt daher, daß die Komponenten von Dateien gewöhnlich Datensätze (record) genannt werden, deren Bestandteile Datenfelder heißen.

12.1 Definition eines Record-Typs

Ein Record-Typ ohne varianten Anteil wird also allgemein wie folgt definiert:

```
type RecordTyp = record s1: Typ1;
                        s2: Typ2;
                        ...
                       sn: Typn
                 end
```

Die durch eine Record-Typdefinition eingeführten Bezeichner `s1, ... , sn` sind die **Variablennamen der einzelnen Komponenten**.

Dieser Record-Datentyp besteht aus n Feldern. Die Felder sind durch Semikolon getrennt. Jedes Feld besteht aus dem Namen der Komponente und dem Datentyp der Komponente, getrennt durch einen Doppelpunkt. Die Namen mehrerer Felder mit demselben Datentyp können zu einer durch Komma getrennten Liste zusammengefaßt werden.

Beispiele für Typdefinitionen von Records sind:

```
type bruch = record zaehler: real;
                    nenner: real
             end
```

(oder abgekürzt: `type komplex = record zaehler, nenner : real end`)

```
type koordinaten = record x : real;
                          y : real
                   end

type datum = record
               tag   : integer;
               monat : integer;
               jahr  : integer;
             end
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
type student   =    record Name       : string [10];
                           Vorname    : string [10];
                           Geburtstag : datum;
                           verheiratet: boolean;
                    end;


var       meyer, krause : student;
          semester      : array [1..50] of student;
```

Anmerkungen:

- `meyer` ist Ganzvariable. Hat vier Komponenten `Name, Vorname, Geburtstag, verheiratet`
- Die Komponenten können von beliebigem Typ sein, also auch selbst wieder ein Record
- Die Definition des Records `datum` muß seiner Verwendung vorausgehen
- Auch die Komponenten eines Array können vom Typ eines Record sein
- Ein Record kann auch Arrays als Komponenten enthalten.
- Alle Feldnamen eines Records müssen verschieden sein
- In verschiedenen Records dürfen Felder gleichen Namens auftreten.

Operationen auf Ganzvariablen:

- Zuweisung
- Auswahl (Selektion von Komponenten)

Zuweisung:

```
person := meyer
```

damit wird der Variablen `person` insgesamt der Wert aller Komponenten der Variablen `meyer` übergeben

Auswahl:

Auf die Felder einer Variablen eines Record-Datentyps kann durch Feldauswahl zugegriffen werden. Sei `d` eine Variable vom Typ `datum`, dann ist `d.Tag` eine Variable vom Typ `integer`, `d.Monat` eine Variable vom Typ `integer`, `d.Jahr` eine Variable vom Typ `integer`.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Wenn eine Ganzvariable x deklariert ist, dann wird auf die Komponente komp mit Hilfe des Record-Bezeichners, dem Selektor-Operator, der durch einen . dargestellt wird, und dem Selektor-Namen (Feldnamen) komp zugegriffen:durch:

```
x.komp
```

Der Selektor besteht also aus Feldbezeichnern und nicht aus errechenbaren Indexwerten wie bei Arrays.

Beispiele:

```
meyer.verheiratet    := false;
meyer.geburtstag.tag := 12;
```

`semester[3].name` ist der Name des 3. Studenten

Operationen auf Komponentenvariablen:

alle Operationen, die für den Komponententyp des Verbundtyps gelten

Abschließend noch ein Beispiel für die Verwendung von Records. Das folgende Beispielprogramm frägt erst die Personendaten eines Schülers ab und kopiert anschließend den gesamten Datensatz in ein Array. Dabei wird nur die **Ganzvariable** verwendet.

```
program klasse;

type person = record vorname, nachname : string[20];
                     alter : Integer;
                     end;

var schueler : person;
    schulklasse : array [1..30] of person;

begin
  write ('Vorname angeben : ');
  readln(schueler.vorname);

  write ('Nachname angeben: ');
  readln(schueler.nachname);

  write ('Alter für ', schueler.vorname , ' : ');
  readln(schueler.alter);

  {record kopieren}
  writeln;
```

```
  writeln (schueler.vorname, ' wird eingeschult...');
  schulklasse[2] := schueler;

  writeln;
  writeln ('Die eingegebenen Werte sind:');
  writeln ('Name : ', schulklasse[2].vorname,' ',
           schulklasse[2].nachname);
  writeln ('Das Alter ist: ', schulklasse[2].alter);
end.
```

12.2 Vergleich Record- und Array-Struktur

Die Record- und Array-Strukturen haben die gemeinsame Eigenschaft, daß beide **beliebig zugreifbare Strukturen (random access structures)** sind. Der **Record** ist insofern allgemeiner, als **nicht alle Komponententypen identisch sein müssen**. Der **Array** seinerseits bietet größere Flexibilität, da die **Komponenten-Selektoren berechenbare Werte** sein können (dargestellt durch Ausdrücke), während die **Selektoren von Record-Komponenten festgelegte**, in der Record-Typen-Definition enthaltene **Namen** sind.

Selbstverständlich kann man auch Arrays von Records erzeugen. Auf einzelne Datenelemente greift man von aussen nach innen zu, beispielsweise:

Tobis_Geburtsjahr := semester[10].geburtstag.jahr;
Anfangsbuchstabe := semester[2].Name[1][38]

[38] Auf einen String kann man wie auf ein Array aus Chars zugreifen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

13 Lektion 13 - Dateien

Informationen im Arbeitsspeicher existieren nur solange, wie das Programm läuft. Nach dem Programmende stehen sie nicht mehr zur Verfügung. Informationen dauerhaft auf Datenträgern speichern zu können, ist der Sinn von Dateien.

Von Pascal aus kann man direkt auf Dateien zu schreiben oder aus ihnen Daten in das Programm einzulesen. [39]

Eine Datei hat einen **Dateinamen**. Unter diesem Namen ist sie dem Betriebssystem des Rechners bekannt und kann durch das entsprechende Betriebssystemkommando wie z.B. `dir` an der Kommando-Schnittstelle des Betriebssystems angezeigt werden.

Aus dem Programm schreibt man in einen Kanal wie z.B. bei der Ausgabe an den Bildschirm nach `output`[40]. Dieser Kanal muss als Dateivariable im Programm vereinbart sein und muss durch die `assign`-Anweisung mit dem Dateinamen gekoppelt werden.

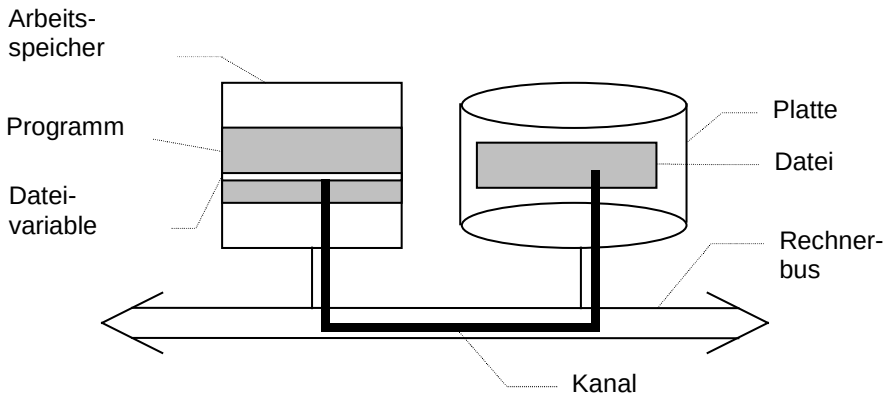


Bild 13-1 Der Kanal zur Platte verbindet das Programm mit der Datei

Eine Datei - nicht jedoch `input` und `output` - muss nach dem `assign` geöffnet werden. In Standard-Pascal kann sie zum Lesen oder zum Schreiben geöffnet werden. Standard-Pascal sieht für Lesen `reset` vor, für Schreiben `rewrite`. Der Schreib/Lesezeiger der Datei wird dabei auf den Anfang gesetzt

Bei Turbo Pascal sind im Gegensatz zu Standard-Pascal nach einer Eröffnung mit `reset` sowohl Lese- als auch Schreibzugriffe erlaubt.

---

[39] Passcal unterstützt sogenannte **sequentielle Dateien**. Dies sind Dateien, bei denen man Informationen nur durch sequentielles Lesen durchsuchen kann. Wenn man also eine bestimmte Information sucht, muss man eine Information nach der anderen lesen, bis man zur richtigen Information gelangt ist.

[40] `output` zeigt standardmäßig auf den Bildschirm, der auch als Datei behandelt wird. Es ist jedoch auch möglich, `output` auch in Dateien auf einem Datenträger umzulenken.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Ist man mit Lesen bzw. Schreiben fertig, so ist die Datei wieder zu schließen. Dazu gibt es die Prozedur `close`.

Pascal kennt zweierlei Dateientypen:

- Textdateien vom Typ `text`
- und allgemeine Dateien

13.1 Schreiben und Lesen für Text-Dateien

Textdateien sind zeilenorientiert wie der Bildschirm und die Tastatur. `input` und `output` sind Dateien vom Typ `text`.

Ein **Dateityp** ist ein **strukturierter Typ**. Er besteht aus einer Folge von **Komponenten des gleichen Typs** (dem **Komponententyp)**. Auf dieser Struktur gibt es eine Position in der Folge der Komponenten, sowie einen Modus, der anzeigt, ob die Datei gelesen oder geschrieben wird.

Bei **Textdateien** schreibt man einen **Zeichenstrom** weg. Die **Struktur** wird erzeugt durch Zeilen. Dabei kann die Folge von Zeichen in **null oder mehr Zeilen** strukturiert sein.

Der vordefinierte Typbezeichner **`text`** stellt einen speziellen Dateityp dar, der dazu dient, um eine Folge von Zeichen aufzunehmen, die in verschieden lange **Zeilen** unterteilt sein kann. Eine **Datei vom Typ `text`** ist also **in null oder mehr Zeilen strukturiert**. Eine Zeile enthält null oder mehr **Zeichen** (Werte vom Typ **`char`**), gefolgt von einem speziellen **Zeilenendezeichen**. Eine Variable vom Typ Text wird **Textdatei** genannt.

Zu beachten ist:

1. Ein Datei-Variable vom Typ **`text`** muß vereinbart werden, ihr Variablenname sei `f`.

   Beispiel: `var  f  : text;`

   Anmerkung: `f` stellt einen logischen Kanal auf die Platte dar.

2. Der Dateiname f ist im Programmkopf als Programmparameter anzugeben

   Beispiel:

```
program FILE_IO (f, input, output);
```

Bei Turbo Pascal kann die runde Klammer mit den Dateien entfallen.

3. Im Programm muß die Verbindung zwischen dem Kanal und dem Dateinamen des Dateisystems des Betriebssystems hergestellt werden. Dies erfolgt mit Hilfe der Standardprozedur **`assign`**, beispielsweise in der Form:

```
assign (f, 'Artikel.Dat')
```

Hier wird dem Kanal `f` die Datei `Artikel.Dat` des Dateisystems zugewiesen.

4. Die Datei ist mit `close(f)` zu schließen

Schreiben:

Mit **rewrite (f)** wird der Dateizeiger auf den Dateianfang positioniert.

Das Schreiben erfolgt mit dem Befehl:

```
write (f, 'Optionaler Text_1', Variable_1 : Formatangabe_1);
write (f, 'Optionaler Text_2', Variable_2 : Formatangabe_2);
...
write (f, 'Optionaler Text_n', Variable_n : Formatangabe_n)
```

Man kann die Formatangabe auch weglassen. Generell findet beim Wegschreiben eine Typkonvertierung von Zahlen in der internen Darstellung in die Dezimaldarstellung statt.

Lesen:

Mit **`reset (f)`** wird der Dateizeiger auf den Dateianfang positioniert.

Das Lesen erfolgt für `integer`-, `real`, `char` und `string`-Variable mit dem Befehl:

```
read (f, Variable)
```

an der jeweiligen Position der Textdatei. Zu beachten ist, daß in die Datei eingefügter **`'Optionaler Text_n'`** überlesen werden muß, damit das `read` in eine Variable an der richtigen Position beginnt (der 'Dateizeiger' muß an der richtigen Stelle stehen).

Beispiel für Schreiben:

```
program schreib (f, input, output);

var datei   : text;
    alter   : integer;
    groesse : real;
    name    : string[30];
    zaehler : integer;
begin

   assign (datei, 'beispiel.dat');
   rewrite (datei);

   for zaehler := 1 to 3 do
   begin
      write ('Bitte Namen angeben: ');
      readln(name);
```

```
      write ('Bitte Alter fuer ',name, ' angeben: ');
      readln(alter);

      write ('Bitte Groesse in Meter fuer ',name, ' angeben: ');
      readln(groesse);
      {Daten in Datei schreiben}
      writeln(datei,alter:6,groesse:8:3,' ',name);
   end;
   close (datei);
end.
```

Beispiel für Lesen:

```
program lese (f, input, output);

var datei   : text;
    alter   : integer;
    groesse : real;
    name    : string[30];

begin
   assign (datei, 'beispiel.dat');
   reset (datei);

   while not eof (datei) do
      begin
         {Datensatz auslesen}
         readln(datei,alter,groesse,name);
         {Ausgabe auf Bildschirm}
         writeln ('Name   :',name);
         writeln ('Alter  :',alter:4);
         writeln ('Groesse:', groesse:6:2);
      end;
   close (datei);
end.
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

13.2 Schreiben und Lesen für allgemeine Dateien

Bei **allgemeinen Dateien** schreibt man **Datensätze unformatiert** weg. Die einzelnen Komponenten werden **unformatiert, d.h. binär**, weggeschrieben, es sei denn es handelt sich um String- oder Character-Variable (char)[41]. Dabei hat **jeder Datensatz dieselbe Struktur**.

Zugriff auf allgemeine Dateien

Man sieht auf die Datei stets durch ein Fenster, das einen Datensatz umfaßt.

Sequentieller Zugriff

Eine sequentielle Datei kann sequentiell geschrieben werden oder sequentiell gelesen werden:

Lesen einer vorhandenen Datei

Nach dem Öffnen der Datei steht der Dateizeiger auf dem ersten Datensatz. Durch sequentielles Lesen kann man einen Datensatz nach dem anderen lesen.

Schreiben einer neuen Datei

Beim Öffnen einer Datei mit `rewrite` steht das Schreibfenster (der Dateizeiger) auf dem ersten Satz, der geschrieben werden kann. Nach dem Schreiben steht das Fenster einen Satz weiter: der zweite Satz kann angelegt werden.

Bei Standard-Pascal ist eine Datei nur im Zustand Schreiben oder nur im Zustand Lesen. Deshalb kann man nicht, wenn man einen Satz an das Ende einer vorhandenen Datei anfügen will, die Datei bis zum Ende lesen, um am Ende einen Satz schreibend hinzuzufügen. In Standard-Pascal muß man die vorhandene Datei Satz für Satz lesen und jeden Satz in eine zweite Datei wegschreiben. Ist man in der Lese-Datei bei der Dateiende-Marke (end of file) angelangt, so weiß man, daß das Ende da ist und hängt in der zu schreibenden Datei den neuen Satz an.

Die Typangabe einer Datei beschreibt ihre Komponenten. Eine Komponente wird auch Datensatz genannt.

Beispiele:

```
type satztyp = record
                    tag   : integer;
                    monat : integer;
                    jahr  : integer
               end;
```

[41] Character und Strings werden im Speicher und auf Platte in der gleichen Form gespeichert, wie man sie am Bildschirm sieht.

```
var        f : file of integer;
           g : file of array [1..8] of real;
     termine : file of satztyp;
```

`f` ist eine Datei, bei der jeder Satz aus 1 `integer`-Zahl besteht.
g ist eine Datei, bei der jeder Datensatz aus 8 `real`-Zahlen besteht.
`termine` ist eine Datei, deren Datensätze aus den Datenfeldern für `Tag`, `Monat` und `Jahr` bestehen.

Schritt1:

Beim Schreiben und Lesen von Records wird empfohlen, nicht direkt mit Typen für Dateien, sondern mit Typnamen für Datensätze und Dateien zu arbeiten. Es soll also ein file-Typnamen und ein Satz-Typnamen spezifiziert werden:

a) Satz-Typnamen spezifizieren. Eine Variable dieses Typs muß dann vereinbart werden; ihr Variablenname sei s

b) File-Typnamen spezifizieren. Eine Variable dieses Filetyps muß dann vereinbart werden. Der Name dieser Dateivariablen sei `f`.

```
type satztyp = record
                  key   : integer;
                  daten : datentyp
               end;

      filetyp   = file of satztyp

 var f:  filetyp;
     s:  satztyp;
```

Diese Struktur kann für jeden Record gelten. Vor der Definition von Satztyp muß jedoch `datentyp` definiert sein. (siehe hierzu das Beispiel unten).

Schritt2:

Der Dateiname f ist im Programmkopf als Programmparameter anzugeben.

Beispiel:

```
program FILE_IO (f, input, output);
```

Schritt3:

Im Programm muß die Verbindung zwischen der logischen Datei und dem Dateinamen des Dateisystems des Betriebssystems hergestellt werden. Dies erfolgt mit Hilfe der Standardprozedur **assign**, beispielsweise in der Form:

```
assign (f, 'Artikel.Dat')
```

Hier wird der logischen Datei `f` die Datei `Artikel.Dat` des Dateisystems zugewiesen.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Schritt4:

Dateizeiger für den Schreib- bzw. den Lesemodus auf den Dateianfang positionieren.

Schritt 5:

Schreiben:

Mit `rewrite(f)` wird der Dateizeiger auf den Dateianfang positioniert.

Das Schreiben erfolgt mit dem Befehl:

```
write(f, s)
```

Lesen:

Mit `reset (f)` wird der Dateizeiger auf den Dateianfang positioniert.

Das Lesen erfolgt mit dem Befehl:

```
read(f, s)
```

6. Die Datei ist mit `close(f)` zu schließen

Beispiel für Schreiben:

```
program schreib(input,output,f);

type artikel = record
                  ...
                 end;

     datentyp = artikel;

     satztyp = record
                  key:    integer;
                  daten:  datentyp
                 end;

     filetyp = file of satztyp;

var        f:         filetyp;
var        s:         satztyp;
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
procedure liessatz (var x:satztyp);    (* liest die Daten des
Satzes vom Terminal ein *)
begin
...
end (* end of liessatz*);

begin
  assign(f,'Artikel');
  rewrite (f);
  liessatz (s);
  while s.key > 0 do
          begin
            write (f,s);
            writeln ( '*** nächster Artikel ***');
            liessatz(s);
          end;
  close(f);

end.
```

Beispiel für Lesen:

```
program lies(input,output,f);

type artikel = record
                        ...
                   end;

     datentyp = artikel;

     satztyp = record
                         key:    integer;
                         daten:  datentyp
                   end;

     filetyp = file of satztyp;

var  f: filetyp;
     s: satztyp;
...

begin
  assign(f,'Artikel');
  reset (f);

  while not eof(f) and (schluessel <>...) do
          begin
             read (f,s);
             if s.key = schluessel then
                  begin
                  ...
```

```
                end;
          end;
  close(f);

end.
```

13.3 Aufgaben

Aufgabe 1 Allgemeine Datei: Split-Programm,

Es kommt häufig vor, dass man Dateien transportieren will, die nicht mehr auf eine Diskette passen, weil sie zu groß sind. Das split-Programm liest große Dateien ein und schreibt sie in kleinen Stücken in .eine Folge von Dateien. Zusätzlich wird eine Batch-Datei generiert, mit der die zerstückelte Datei wieder zusammengesetzt werden kann.

Das Grundgerüst für das Programm ist schon fertig. Fügen sie die noch benötigten Anweisungen hinzu.

```
program split;
uses crt;

type bytedatei = file of char;

var inputdatei, outputdatei : bytedatei; { Dateien }
    zeichen : char;
    inzaehler, outzaehler: longInt;
    dateizaehler    : integer;

    neueDatei : boolean;
    maximal_groesse : longInt;

    inputdateiname,pfad : String;
    outputdateiname : string;

    batchstring : string;

{ Die Prozedur neueOutputDatei erzeugt einen Dateinamen
  fuer die gesplittete Datei. Dann wird die Datei geoeffnet.}
  procedure neueOutputDatei(position : integer;
                            pfad : string;
                            outputdateiname: string;
                            var datei : bytedatei;
                            var batch_string: string);
    var zahl_wort : string[3];
        dateiname,name : string;
        punkt_position : integer;
        tempstring : string;
    begin
      {Die Funktion str wandelt eine Zahl in einen String.
       Damit wird aus der Positionsnummer ein String mit der
       Zahl.}
```

```
      str(position, zahl_wort);
      name := concat (outputdateiname,'.');
      {Hier wird der neue Dateiname aus Pfad, Dateiname und der
       Positionsnummer zusammengesetzt}
      dateiname := concat (pfad, name,zahl_wort);

      {Die neue Datei wird geoeffnet und der Dateizeiger fuer
       Schreiben an den Anfang gesetzt}
      assign (datei , dateiname);

      {todo: Dateizaehler für Schreiben zurücksetzen}
      ...

      {Alle neuen Dateien werden an den batch-String angehaengt.}
      tempstring := concat (batch_string,name,zahl_wort,'+');
      batch_string := tempstring;
    end;

    {In der Prozedur Dateiname_Einlesen werden die Daten fuer
     die Zerhackung eingelesen.
    }
    procedure dateiname_einlesen (var indateiname : string;
                                  var outdateiname : string ;
                                  var groesse : longint);
    const MAXGROESSE = 20000;
    var dateigroesse : longint;
    begin
      clrscr;
      writeln ('    Split');
      writeln ('Das Programm zum Zerhacken grosser Dateien');
      writeln;
      writeln ('Bitte geben Sie den Pfad und Dateinamen der
Eingangsdatei an');
      readln (indateiname);
      writeln;
      writeln ('Geben Sie den Dateiname (ohne Pfad und
Erweiterung) fuer die Ausgangsdatei an:');
      flush(output);
      readln(outdateiname);
      repeat
        write ('Geben Sie die Groesse der zerhackten
Ausgangsdateien an (in Kilobyte) : ');
        flush (output);
        read (dateigroesse);
      until ((dateigroesse > 0) and (dateigroesse < MAXGROESSE));
      groesse := dateigroesse * 1024; {Umwandlung in Kilobyte}
    end;

    {Diese Prozedur bestimmt den Pfad aus dem Dateinamen.}
    procedure pfad_bestimmen (dateiname : string;
                              var pfad : string);
    var position, zaehler : integer;
```

```
    begin
      position := 0;
      {Die Position des letzten "\" wird ermittelt}
      for zaehler := 1 to length (dateiname) do
        if dateiname [zaehler] = '\' then
          position := zaehler;
      pfad := copy (dateiname, 1, position);
      writeln (pfad);
    end;

    {Die Prozedur erzeuge_BatchDatei erzeugt eine Batchdatei,
     mit der die zerhackte Datei aus den Einzelstuecken wieder
     hergestellt werden kann.}
    procedure erzeuge_BatchDatei (inputdateiname: string;
                                  pfad : string;
                                  outputdateiname : string;
                                  batch_string : string);
    var dateiname: string;
        indateiname : String;
        batchdatei : text;
        textstring : string;

    begin
     {Den Namen (ohne Pfad) der Eingangsdatei bestimmen}
     indateiname := copy (inputdateiname, length (pfad)+1,
                          length (inputdateiname)-length(pfad));
     {Das letzte Zeichen des batchstrings entfernen}
     delete (batch_string,length (batch_string),1);
     textstring := concat (batch_string,' ',indateiname);

     {Textdatei erzeugen, in die die Anweisungen
      geschrieben werden}
     dateiname := concat (pfad,outputdateiname,'.bat');

     {todo: textdatei batchdatei mit dateiname verbinden}
     assign (..., ...);

     {todo: batchdatei für schreibenden Zugriff zurücksetzen}
     ...(batchdatei);

     {todo: Batchdatei schreiben}
     writeln (..., 'rem Batchdatei fuer Split');
     writeln (..., textstring);

     {todo: batchdatei schließen}
     close(...);
    end;

{Hauptprogramm}
begin
  dateiname_einlesen (inputdateiname,
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
                    outputdateiname,
                    maximal_groesse);

batchstring := 'copy /b ';
pfad_bestimmen (inputdateiname, pfad);

{todo: Input-Datei vorbereiten für inputdatei mit
       inputdateiname}
...
inzaehler := 0;

{todo: Inputdatei zurückseten für Lesen}
...

{Output-Datei vorbereiten }
dateizaehler := 1;
outzaehler := 0;
neueOutputDatei (dateizaehler ,
                 pfad,
                 outputdateiname,
                 outputdatei,
                 batchstring);
{Dateien kopieren}
while not eof (inputdatei) do
begin
  inzaehler := inzaehler + 1;
  outzaehler := outzaehler + 1;
  {byteweise Datei kopieren}
  {todo: ein Zeichen aus Inputdatei auslesen...}
  read(...,zeichen);
  {todo: ... und in outputdatei schreiben}
   ...

  {wenn die outdatei voll ist, wird sie geschlossen und eine
  neue Datei wird angefangen.}
  if outzaehler >= maximal_groesse then
    begin
      {todo: outputdatei schließen}
      ...

      dateizaehler := dateizaehler +1;
      neueOutputDatei (dateizaehler ,
                       pfad ,
                       outputdateiname,
                       outputdatei,
                       batchstring);
      outzaehler := 0;
    end; {of if}

  {Position der Kopierens anzeigen (alle 10 kbyte)}
  if (inzaehler mod 10240) = 0 then
    writeln (inzaehler div 1024 , ' KB');
```

```
  end; {of while}
{todo: inputdatei schließen}
  ...
{todo: outputdatei schließen}
  ...
{Batchdatei erzeugen}
  erzeuge_BatchDatei(inputdateiname,
                     pfad,
                     outputdateiname,
                     batchstring);

  writeln ('Fertig, es wurden ',dateizaehler, ' Dateien
geschrieben .');
end.
```

Aufgabe 2 Allgemeine Dateien und Textdateien und Sortieren im Arbeitsspeicher

Das folgende Programm erzeugt ein Array von integer-Zufallszahlen. Die Größe des Arrays (<= 1000) wird in der Prozedur eingabe im Dialog eingegeben. Die von der Prozedur `zufallszahlen` erzeugten Zufallszahlen werden von der Prozedur `schreib` in eine allgemeine Datei `f` ausgegeben. Die Prozedur `wandel` liest alle Datensätze der allgemeinen Datei `f` und schreibt die Zufallszahlen mit einer Breite von 10 Schreibstellen (Zeichen) in eine Textdatei g. Die Prozedur `sort` liest alle Zufallszahlen von g in den Arbeitsspeicher ein und sortiert sie im Arbeitsspeicher nach der Methode des direkten Aussuchens (straight selection) und schreibt die sortierten Zufallszahlen mit einer Stellenbreite von 10 Zeichen in die Textdatei h weg.

```
program sortieren (input, output);

const   maximal      =     1000;

type   zahlenfeld    =     array [1..maximal] of integer;

       filetyp       =     file of integer;

var  anzahl    :    integer;
       f       :    filetyp;
       g       :    text;
       h       :    text;
       A       :    zahlenfeld;

procedure eingabe (var z : integer);

begin
   writeln ('Geben Sie die Anzahl der zu generierenden');
   writeln ('Zufallszahlen ein [maximal 1000]: ');
   read (z);
end;

procedure zufallszahlen (.....);

const   a     =  109;
```

```
        b     =  853;
        m     = 4096;
        c     =   26;
var LV        :     integer;

begin
   zufall [1]  := c;
   for LV:= 2 to x do
       zufall [LV] := (a*zufall[LV-1] + b) mod m;
end;


procedure schreib ( var x : filetyp; y : integer;
                    z : zahlenfeld);
var LV : integer;

begin
   for LV := 1 to y do
      write (x, z[LV]);
end;


procedure wandel (.....);

var zahl : integer;

begin
    .....
end;

procedure sort (var x: text; var y: text);

var j      : integer;
    k      : integer;
    zahl   : zahlenfeld;
    index  : integer;    (* index von zahl *)
    max    : integer;
    min    : integer;

begin
            (* Zahlen in Array zahl [index] des Arbeits- *)
            (* speichers einlesen                        *)

   writeln ('Die Anzahl der Zahlen ist: ', max);

   for index := 1 to max - 1 do
       begin
          k   := index;
          min := zahl[index];

        ..... (* kleinstes Element im unsortierten Teil    *)
              (* suchen und feststellen, ob zahl[index]    *)
```

```
                (* oder das kleinste Element im unsortierten *)
                (* Teil das Minimum darstellt                *)
                (* Stellt das kleinste Element im sortierten *)
                (* Teil das Minimum dar, so wird dieses      *)
                (* Element mit zahl (index) vertauscht       *)

   for index:= 1 to max do           (* wegschreiben der    *)
      writeln (h, zahl[index]:10);   (* sortierten Zahlen   *)
end;

begin                                      (* Hauptprogramm *)
    assign (f, 'ZUFALL.DAT');
    assign (g, 'ZUFALL.TXT');
    assign (h, 'ZUFALL.SOR');
    eingabe (anzahl);
    zufallszahlen (anzahl, A);
    rewrite (f);
    schreib (f, anzahl, A);
    reset (f);
    rewrite (g);
    wandel (f,g);
    close (f);
    reset (g);
    rewrite (h);
    sort (g,h);
    close(g);
    close(h);
end.
```

2.1 Ergänzen Sie die Parameterliste der Prozedur zufallszahlen.
2.2 Schreiben Sie die Prozedur `wandel`.
2.3 Ergänzen Sie die fehlenden Teile der Prozedur `sort`.
2.4 Erstellen Sie das Struktogramm (Nassi-Shneiderman-Diagramm) für das Hauptprogramm und die Prozedur sort.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Aufgabe 3: Arrays, Records, Textdateien

Das folgende Programm `messprotokoll` erlaubt es, Meßdaten, die in einem Array a stehen, in ein Array b nach der Größe der Meßwerte zu sortieren und das ursprüngliche Array a in die Textdatei `LOG.TXT`, das sortierte Array b in die Datei `SORT.TXT` zu schreiben.

Die Meßwerte sind Werte, die zu jeder vollen Stunde gemessen werden. Das Array a bzw. das Array b ist ein Array aus 24 Records, wobei jeder Record den Meßwert zu einer vollen Stunde und die jeweilige Stunde selbst enthält :

```
const    unten   = 1;
         oben    = 24;

type messdaten  = record
                     stunde : integer;
                     wert   : real;
                  end;
/* stunde ist eine Zahl zwischen 1 und 24, entspricht den */
/* vollen Stunden                                         */
/* wert ist der Meßwert zur jeweiligen Stunde             */

     feld = array [unten..oben] of messdaten;

var a, b : feld;
```

Das Sortieren erfolgt so, daß im Array b die Meßwerte in aufsteigender Reihenfolge vorliegen, d.h. in der Array-Komponente `b[1]` soll der Record mit dem kleinsten Meßwert und in der Array-Komponente `b[24]` soll der Record mit dem größten Meßwert stehen.

`a[1]` enthält den Meßwert um 1 Uhr b [1] enthält den kleinsten Meßwert
`a[2]` enthält den Meßwert um 2 Uhr.....
.....
`a[24]` enthält den Meßwert um 24 Uhr b[24] enthält den größten Meßwert

Beispiel:

volle Stunde	**1**	**2**	**3**	**4**	**5**	**.....**
Meßwert	**2.5**	**2.8**	**2.4**	**3.1**	**2.**	**.....**

a[1] a[2] .....

Die Prozedur **`eingabe`** dient zum Eingeben der Meßwerte, die Prozedur **`sort`** zum Sortieren und die Prozedur **`textwrit`**e zum Erzeugen der beiden gewünschten Textdateien, **`LOG.TXT`** und **`SORT.TXT`**.

Fehlende Teile des Programms sind durch **`.....`** gekennzeichnet.

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
program messprotokoll (f, g, input, output);

const unten = 1;
      oben  = 24;

type messdaten = record
                    stunde: integer;
                    wert  : real;
                 end;

     feld = array [unten..oben] of messdaten;

var  a, b  : feld;
     f ,g  : text;


procedure eingabe (. . . . .); /* Ergänzen Sie die fehlenden Teile */

var lv : integer;

begin
   . . . . .
/* Ergänzen Sie die fehlenden Teile */
end;


procedure sort (m: feld; var n: feld);

var  lv         : integer;    /* Laufvariable                    */
     hilf       : messdaten;  /* Hilfsvariable für das Tauschen  */
     obergrenze : integer;

begin
   obergrenze := oben;
   /* äußere Schleife           */
   while obergrenze > unten do
       begin
         for lv:= unten + 1 to obergrenze do /* innere Schleife */
            . . . . ./* Ist der Meßwert des Vorgängers größer   */
                     /* als der Meßwert des jeweiligen          */
                     /* Array-Elements, so werden die beiden    */
                     /* Array-Elemente vertauscht               */
                     /*Ende der inneren Schleife                */
         obergrenze := obergrenze - 1
       end;
   /* Ende der while-Schleife           */

   . . . . .      /* Array m in Array n kopieren             */

end;

/* Anmerkung:                                                 */
/* Nach dem 1. Durchlauf durch die innere Schleife steht.....*/
/* das Array-Element mit dem größten Meßwert am Platz mit     */
/* dem größten Index. Nun muß nur noch der Restsortiert       */
/* werden. Das geschieht mit demselben Algorithmus, aber mit  */
/* einer um 1 verringerten Obergrenze, solange, bis die       */
/* Obergrenze 1 erreicht.                                     */
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
procedure textwrite;

var lv: integer;

begin
   . . . . .   /* Ergänzen Sie die fehlenden Teile */
end;


begin
   writeln;
   writeln ('Initialisierung von a');
   eingabe (a);
   writeln ('Nun wird sortiert');
   sort (a,b);
   textwrite;
end.
```

3.1 Ergänzen Sie die fehlenden Teile der Prozedur **`eingabe`**. Hierbei sollen im Dialog die 24 Meßwerte von 1 Uhr bis 24 Uhr in das Array a eingegeben werden. Um welche Stunde es sich jeweils handelt, ist beim Abfragen nach dem zugehörigen Meßwert im prompt anzuzeigen. Die angezeigte Stunde und der zugehörige eingegebene Meßwert sind in die jeweilige Arraykomponente von a einzutragen.

3.2 Ergänzen Sie die Prozedur **`sort`** gemäß dem Algorithmus, der als Kommentar angegeben ist. In der Arraykomponente von b mit dem kleinsten Feldindex soll die Messung mit dem kleinsten Meßwert stehen.

3.3 Schreiben Sie die Prozedur `textwrite`. Die Textdatei **`LOG.TXT`** (Array a weggeschrieben in `LOG.TXT`) soll das folgende Aussehen haben (das folgende Beispiel hat nicht 24, sondern nur 5 Stunden):

```
Stunde:┘       1┘ Wert:┘   9.00
Stunde:┘       2┘ Wert:┘  17.00
Stunde:┘       3┘ Wert:┘   2.00
Stunde:┘       4┘ Wert:┘   5.00
Stunde:┘       5┘ Wert:┘  16.00
```

┘ stellt ein blank dar. Die Stunde ist mit einer Stellenwertbreite von 2 Stellen, der Meßwert mit 2 Stellen vor dem Komma und 2 nach dem Komma auszugeben.

Die Datei **`SORT.TXT`** (Array b weggeschrieben in `SORT.TXT`) soll dasselbe Format haben. Beispiel:

```
Stunde:┘       3┘ Wert:┘   2.00
Stunde:┘       4┘ Wert:┘   5.00
Stunde:┘       1┘ Wert:┘   9.00
Stunde:┘       5┘ Wert:┘ 16.00
Stunde:┘       2┘ Wert:┘ 17.00
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

Aufgabe 4: Allgemeine Dateien

Das folgende Programm `datei_vereinigung` dient zum Vereinigen zweier existierender allgemeiner Dateien `PREIS.DAT` und `STUECK.DAT` in eine allgemeine Datei `GESAMT.DAT`. Der Aufbau der Dateien ist aus den Typdefinitionen des Programms ersichtlich.

Das folgende Bild 13-2 dient nur der Veranschaulichung. Selbstverständlich sind die Felder vom Typ integer und real binär gespeichert.

Datei: `PREIS.DAT`

Artikel-Nr.	Artikelname	Preis
1	Lineal	1.2
2	Heft	2.5
.	.	.
.	.	.

`STUECK.DAT`

Artikel-Nr.	Artikelname	Stueckzahl
1	Lineal	10
2	Heft	14
.	.	.
.	.	.

`GESAMT.DAT`

Artikel-Nr.	Artikelname	Preis	Stueckzahl
1	Lineal	1.2	10
2	Heft	2.5	14

Bild 13-2 Veranschaulichung der Struktur der 3 allgemeinen Dateien

Jeweils der n. Satz von `PREIS.DAT` und von `STUECK.DAT` enthält Informationen über denselben Artikel. Die Artikelnummer und der Artikelname sind dabei redundant in den zwei Dateien `PREIS.DAT` und `STUECK.DAT` gespeichert. Es soll nun eine Datei `GESAMT.DAT` erzeugt werden, die in ihrem n-ten Satz alle Informationen des n-ten Satzes von `PREIS.DAT` und des n-ten Satzes von `STUECK.DAT` enthält.

Ergänzen Sie die fehlenden Teile der Prozedur `mergen`

```
program dateimerge (f,g,h,input, output);

type auswahl = string [5];

     satz1 = record
                 Artikelnummer : integer;
                 Artikelname   : string[20];
                 Preis         : real;
             end;

     satz2 = record
                 Artikelnummer : integer;
                 Artikelname   : string[20];
                 Stueckzahl    : integer;
             end;
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000

```
     satz3 = record
                 Artikelnummer : integer;
                 Artikelname   : string[20];
                 Preis         : real;
                 Stueckzahl    : integer;
             end;

var f      : file of satz1;
    g      : file of satz2;
    h      : file of satz3;
    ende   : boolean;
    wahl   : auswahl;


procedure datei_gen;

var u                      : satz1;
    v                      : satz2;
    Artikelname            : string [20];
    lv                     : integer;

begin
   assign (f,'PREIS.DAT');
   assign (g, 'STUECK.DAT');
   rewrite (f);
   rewrite (g);
   for lv := 1 to 3 do
     begin
        Artikelname := '                    ';
        u.Artikelnummer := lv;
        v.Artikelnummer := lv;
        writeln ('Gib den Artikelnamen ein: ');
        readln (Artikelname);
        u.Artikelname := Artikelname;
        v.Artikelname := Artikelname;
        writeln ('Gib den Preis ein: ');
        readln (u.Preis);
        writeln ('Gib die Stueckzahl ein: ');
        readln (v.Stueckzahl);
        write (f,u);
        write (g,v);
     end;
   close(f);
   close (g);
end;

procedure mergen;

var u: satz1;
    v: satz2;
    w: satz3;
```

```
begin
    assign (f,'PREIS.DAT');
    assign (g,'STUECK.DAT');
    assign (h,'GESAMT.DAT');
    reset (f);
    reset (g);
    rewrite (h);
    while not eof (f) do
      begin
          . . . . .
      end;
    close (f); close (g); close (h);
end;


procedure menue (var x: auswahl);

begin
   writeln ('Zur Auswahl stehen: ');
   writeln ('Dateien PREIS.DAT und STUECK.DAT');
   writeln ('generieren                          [dgen]');
   writeln ('Gesamtdatei GESAMT.DAT aus Dateien');
   writeln ('PREIS.DAT und STUECK.DAT generieren [gesa]');
   writeln ('Programm beenden                    [stop]');
   write   ('Eingabe:                            ');
   readln (x);
end;


begin
   ende := false;
   while (ende = false) do begin
     menue (wahl);
     if wahl = 'dgen' then datei_gen
     else if wahl = 'gesa' then mergen
     else if wahl = 'stop' then ende := true;
   end;
end.
```

© Joachim Goll/Johannes-Markus Waizenegger, STZ Softwaretechnik Version: 09.09. 2000